TURING 图灵程序设计丛书 微软技术系列

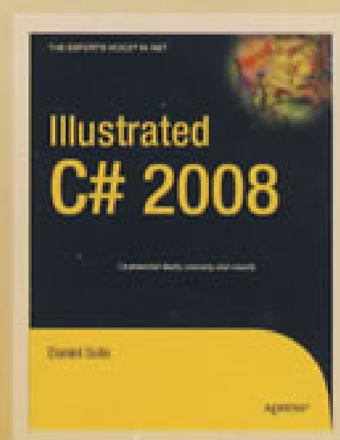
Apress®

# Illustrated C# 2008

# C#图解教程

[美] Daniel Solis 著

苏林 朱晔 等译




- 用图说话，最易学的C#教程
- Amazon全五星盛誉
- 涵盖Visual C# 2008和.NET 3.5最新特性

人民邮电出版社
POSTS & TELECOM PRESS

# 版 权 声 明

谨将此书献给我的父母——Salt和Amy，并献给Sian和Sue。

# 译者序（一）

书是知识的载体，是智慧的传播者。技术图书在技术的普及、发展过程中的作用是毋庸置疑的。在这个知识爆炸、信息技术迅猛发展的时代，技术图书的作用更加突出。我们比以往任何时候都需要关于新技术和新平台的参考资料。一本描述清晰、内容详细的书能使我们快速掌握这些技术。

译者不才，自己无力写出这样的书，愿意以虫蚁之能，行搬运之事，将优秀外文书籍译成中文，以利于国人参考和学习，从而为技术传播尽自己的绵薄之力。

C#和.NET平台近年来迅速普及，已经成为很多公司使用的主要技术之一。有很多出色的应用都是使用C#开发的，包括很多Web 2.0时代的网络应用。虽然.NET平台目前还只能在Windows操作系统下工作，但是这并没有妨碍它发展壮大。一方面是因为Windows操作系统的普及程度已经给.NET提供了巨大的发展空间；另一方面是因为.NET确实是个优秀的平台，而且C#也确实算得上是新一代的优秀的面向对象编程语言。作为一个与时俱进的软件工程师，忽视C#和.NET是很不明智的。

本书是一部极为出色的C#著作。正如本书作者所说，它不仅包含了入门的基础知识，而且同时还能作为开发过程中的参考书使用。书中使用了大量的示例和图表，使内容一目了然。即便是有经验的C#程序员，阅读这本书也会受益匪浅。

在本书的翻译过程中，我尽量保持原书清晰明了的风格，并努力保证术语及用词的准确。由于能力有限，我虽已尽所能，但仍难免有不妥之处，望读者朋友海涵。

感谢我的妻子毛毛！在我翻译本书的过程中，她承担了大部分的家务，并给予了我很多支持和鼓励。没有她的爱和付出，本书的翻译工作肯定不会进展得如此顺利。

相信这本书一定对你有用！

苏　林

2008年5月于上海

# 译者序（二）

.NET经过近8年的发展后已经变得非常庞大，也非常成熟了，而且发展的速度越来越快。

学好C#是在.NET平台上构建一切应用的前提，因此，我强烈建议欲涉足.NET的初学者无论如何都应该学习C#，并且要学好。

说点题外话，很多人没有将.NET框架、CLR和C#语言这三者之间的关系区分清楚，认为其版本号是一一对应的。其实，.NET框架是一个独立发布的软件包，包括了CLR、类库以及相关的语言编译器等工具。C#代码经过编译之后在CLR环境中运行。由于.NET 3.0/3.5其实是.NET 2.0的扩展（只是增加了一些新的程序集），所以.NET 3.0/3.5的CLR版本还是2.0。而且，.NET 3.0其实只扩展了WF、WPF、WCF、WCS等组件，并没有提供新的C#编译器，直到.NET 3.5中才打包了C#3.0的编译器。所以，.NET框架、CLR和C#的版本之间的对应关系如下表所示：

| .NET版本 | 1.0 | 1.1 | 2.0 | 3.0 | 3.5 |
|---|---|---|---|---|---|
| CLR版本 | 1.0 | 1.1 | 2.0 | 2.0 | 2.0 |
| C#版本 | 1.0 | 1.1 | 2.0 | 2.0 | 3.0 |

也就是说，对于那些不涉及新程序集的C# 3.0新特性（比如自动属性、匿名类型等）在.NET 2.0的环境中也可以运行，CLR对这些特性是一无所知的。

言归正传，拿到本书的英文版后，我粗略地看了一下目录，认为此书是一本彻底面向初学者的基础书籍。在翻译了几章之后，才发现先前的认识不完全正确。此书和一般的面向初学者的C#书籍不太一样：

- 可能和作者的C++背景有关，作者喜欢从底层（比如内存布局）的角度来剖析一些知识点。这有助于读者在知其然的同时还能知其所以然，从而打下扎实的基础。
- 书如其名，本书的特点就是有大量示意性的表格和插图，简洁明了，非常易于读者对知识点的理解。书中还有大量的范例代码，代码中也添加了很多注解，可以帮助读者理解代码的要点。
- 另外，本书绝对不是老版的旧酒装新瓶。C# 3.0的所有新特性都完全地融合在其中，而不是在老版本基础上加一些关于新特性的章节。

因此，如果你确实已经使用C#构建了很多应用或已经对C# 2.0有所掌握，那么本书或许会对你非常有用，很多关于CLR本质的内容将能帮助你更深入地理解C#。

由于时间关系，译者在翻译的过程中难免有疏漏。本书的第1~13章由苏林先生翻译，第14~25章由我翻译。欢迎对C#或.NET感兴趣的朋友与我交流，我的邮箱是yzhu@live.com，个人BLOG是 http://lovecherry.cnblogs.com。

最后，预祝你在阅读本书之后能有所收获，编程快乐。

朱　晔

2008年5月于上海

# 前 言

本书的目的是讲授C#编程语言的基础知识和工作原理。大多数图书主要使用文字讲授编程。文字对于小说来说足够了，但对于编程语言中的很多重要概念，综合运用文字、图形和表格会更容易理解。

我们中许多人都习惯于形象思维，而图形和表格有助于我们更清晰地理解概念。在几年的编程语言教学过程中，我发现是我在白板上画的图帮助学生最快地理解了我要传达的概念。

然而，单是图表并不足以解释一种编程语言和平台。本书的目标是找到文字和图表的最佳结合，以使你对这种语言有透彻的理解，并且也让本书能当作参考工具使用。

本书写给所有想要学习C#的人——从初学者到有经验的程序员。刚开始学编程的人会发现，书中全面讲述了基础知识；有经验的程序员会觉得，内容的叙述非常简洁，无需苦苦寻觅就能直接获得想要的信息。对于这两类程序员，内容本身都用图形化方式呈现，这种方式使这种语言更容易学习。

请享受本书吧!

## 致谢

我想感谢Sian每天支持并鼓励我，我还想感谢我的父母、兄弟和姐妹，他们一直爱我并支持我。

我还想对Apress的朋友表达诚挚的感谢，他们与我一起工作并完成这本书。我真心感激他们理解并赏识我努力做的事情，并和我一起完成它。感谢你们所有人。

# 目　录

# 第 1 章 C#和.NET框架

**本章内容**

- 在.NET之前
- 进入Microsoft .NET
- 编译成CIL
- 编译成本机代码并执行
- CLR
- CLI
- 缩写回顾

## 1.1 在.NET 之前

C#编程语言是为开发微软公司的.NET框架[①]上的程序而设计的。本章将简要介绍.NET从何而来，以及它的基本架构。这只是为了确保你从正确的一步开始，让我借此机会提醒你一件可能显而易见的事情：C#的发音为*see sharp*[②]。

### 1.1.1 20 世纪 90 年代后期的 Windows 编程

在20世纪90年代后期，使用微软平台的Windows编程分化成许多分支。大多数程序员在使用Visual Basic（VB）、C或C++。一些C和C++程序员在使用纯Win32 API，但大多数人在使用MFC（Microsoft Foundation Classes，微软基础类库）。其他人已经转向了COM（Component Object Model，组件对象模型）。

所有这些技术都有自己的问题。纯Win32 API不是面向对象的，而且使用它的工作量比使用MFC的更大。MFC是面向对象的，但是它却不一致，并逐渐变得陈旧。COM虽然概念上简单，但它的实际代码复杂，并且需要很多丑陋的、不雅的底层基础代码。

所有这些编程技术的另外一个缺点是它们主要针对桌面程序而不是Internet的开发。那时，Web编程还是以后的事情，而且看起来和桌面编程非常不同。

---

① 微软正式中文文献中一般称.NET Framework，本书考虑了国内读者习惯，统一译为.NET框架。——编者注

② 有一次我去应聘一个C#编程的职位，当时人力资源面试官问我从事"see pound"（应为see sharp）的经验有多少！我过了一会儿才弄清楚他在说什么。

### 1.1.2　下一代平台的目标

我们真正需要的是一个新的开始——一个集成的、面向对象的开发框架，它可以把一致和优雅带回编程。为满足这个需求，微软宣布开发一个代码执行环境和一个可以实现这些目标的代码开发环境。这些目标列在图1-1中。

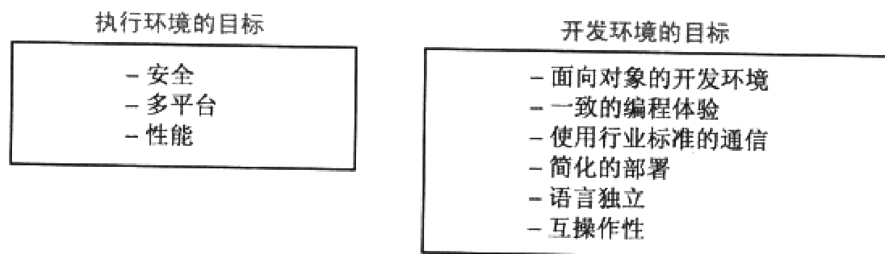



图1-1　下一代平台的目标

## 1.2　进入 Microsoft .NET

在2002年，微软发布了.NET框架，声称其解决了旧问题并实现了下一代系统的目标。.NET框架是一种比MFC或COM编程技术更一致并面向对象的环境。它的特点包括以下几点。

- 多平台：该系统可以在广泛的计算机上运行，包括从服务器、桌面机到PDA和移动电话。
- 行业标准：该系统使用行业标准的通信协议，比如XML、HTTP、SOAP和WSDL。
- 安全性：该系统能提供更加安全的执行环境，即使有来源可疑的代码存在。

### 1.2.1　.NET 框架的组成

.NET框架由三部分组成，如图1-2所示。[①]

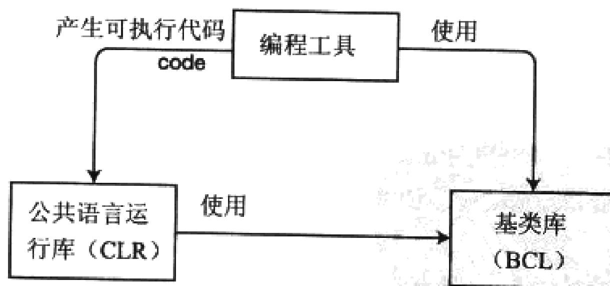



图1-2　.NET框架的组成

执行环境称为CLR（Common Language Runtime，公共语言运行库）。CLR在运行期管理程序的执行，包括以下内容。

- 内存管理

① 严格地说，.NET框架由两部分组成：CLR和FCL（框架类库），不包括工具。FCL是BCL的超集，还包括Windows Forms、ASP.NET LINQ以及更多命名空间。——编者注

- ❑ 代码安全验证
- ❑ 代码执行
- ❑ 垃圾收集

编程工具涵盖了编码和调试需要的一切，包括：

- ❑ Visual Studio集成开发环境
- ❑ .NET兼容的编译器（例如：C#、VB、JScript和托管的C++）
- ❑ 调试器
- ❑ 服务器端改进，比如ASP.NET

BCL（Base Class Library，基类库）是.NET框架使用的一个大的类库，而且也可以在你的程序中使用。

### 1.2.2 大大改进的编程环境

较之以前的Windows编程环境，.NET框架为程序员带来了相当大的改进。下面的几节将简要阐述它的特点及其带来的好处。

**1. 面向对象的开发环境**

CLR、BCL和C#被设计得完全面向对象，并形成良好的集成环境。

系统为本地程序和分布式系统都提供了一致的、面向对象的编程模型。它还为桌面应用程序、移动应用程序和Web开发提供了软件开发接口，涉及的对象范围很广，从计算机服务器到手机。

**2. 自动垃圾收集**

CLR有一项服务称为GC（Garbage Collector，垃圾收集），它能为你自动管理内存。

- ❑ GC自动从内存中删除程序不再访问的对象。
- ❑ GC使程序员不再操心许多以前必须执行的任务，比如释放内存和检查内存泄漏。这可不是个小特性，因为检查内存泄漏可能非常困难而且耗时。

**3. 互操作性**

.NET框架的设计专门考虑了不同的.NET语言、操作系统或Win32 DLL和COM之间的互操作性。

- ❑ .NET语言的互操作性允许在用不同的.NET语言编写的软件模块间无缝地交互。
  - ■ 一种.NET语言写的程序可以使用甚至继承用另外一种.NET语言写的类，只需要遵循一定的规则即可。
  - ■ 正因为它能够很容易地集成不同编程语言生成的模块，.NET框架有时被称为语言无关的。
- ❑ .NET提供一种称为平台调用（platform invoke, P/Invoke）的特性，允许.NET的代码调用并使用非.NET的、但通过标准Win32 DLL导出的纯C函数的代码，比如Windows API。
- ❑ .NET框架还允许与COM的互操作。.NET软件组件能调用COM组件，而且COM也能调用.NET组件，就像它们是COM组件一样。

**4. 不需要COM**

.NET框架使程序员摆脱了COM的束缚。作为一个C#程序员，不需要使用COM，因而也不需要下面这些内容。

- IUnknown接口：在COM中，所有对象必须实现IUnknown接口。相反，所有.NET对象都继承一个名为object的类。接口编程仍是.NET中的一个重要部分，但不再是中心主题了。
- 类型库：在COM中，类型信息保存在类型库中当作.tlb文件，它和可执行代码是分开的。在.NET中，程序的类型信息和代码一起被保存在程序文件中。
- 引用计数：在COM中，程序员必须记录一个对象的引用数目以确保它不会在错误的时间被删除。在.NET中，GC记录引用情况并只在合适的时候删除对象。
- HRESULT：COM使用HRESULT数据类型返回运行时错误代码。.NET不使用HRESULT。相反，所有意外的运行时错误都产生异常。
- 注册表：COM应用必须在系统注册表中注册。注册表保存了与操作系统的配置和应用程序有关的信息。.NET应用不使用注册表，这简化了程序的安装和卸载。（但是有类似的东西，称为全局程序集缓存，即GAC，我会在第10章阐述。）

**5. 简化的部署**

部署为.NET框架编写的程序比以前容易很多，这是由于以下几点原因：

- .NET程序不需要使用注册表注册，这意味着在最简单的情形下，一个程序只需要被复制到目标机器上便可以运行。
- .NET提供一种称为并行执行的特性，允许一个DLL的不同版本在同一台机器上存在。这意味着每个可执行程序都可以访问程序生成时使用的那个版本的DLL。

**6. 类型安全性**

CLR检查并确保参数及其他数据对象的类型安全，即使是在不同编程语言编写的组件之间。

**7. 基类库**

.NET框架提供了一个广泛的基础类库，很自然地，它被称为基类库（Base Class Library，BCL）。（有时称为框架类库——Framework Class Library，FCL。）[①]在写自己的程序时，可以使用这些丰富的代码。包括以下一些类。

- 通用基础类：这些类提供了一组极为强大的工具，可以应用到广泛的编程任务中，比如字符串操作、安全和加密。
- 集合类：这些类实现了列表、字典、散列表以及位数组。
- 线程和同步类：这些类用于创建多线程程序。
- XML类：这些类用于创建、读取以及操作XML文档。

## 1.3 编译成 CIL

.NET语言的编译器接受源代码文件，并生成名为程序集的输出文件。程序集可以是可执行文件或DLL。图1-3阐述了这个过程。

- 程序集里的代码并不是本机代码，而是一种名称为CIL（Common Intermediate Language，公共中间语言）的中间语言。

---

① 严格地说，BCL并不等同于FCL，而只是FCL的一个子集，包括System，System.IO，System.Resources，System.Text等FCL中比较底层和通用的功能。——编者注

- 程序集包含的信息中，包括下列项目：
  - 程序的CIL；
  - 程序中使用的类型的元数据；
  - 对其他程序集引用的元数据。

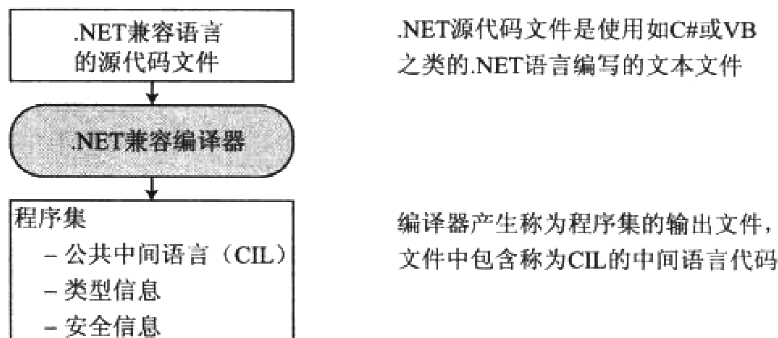



图1-3 编译过程

随着时间的推移，公共中间语言的缩写已经改变，而且不同的参考书可能使用不同的术语。可能会遇到的其他两个CIL的术语是IL（Intermediate Language）和MSIL（Microsoft Intermediate Language），它们在初始发展阶段和早期文档中使用过。

## 1.4 编译成本机代码并执行

程序的CIL直到它被调用运行时才会被编译成本机代码。在运行时，CLR执行下面的步骤（如图1-4所示）：

(1) 检查程序集的安全特性；

(2) 在内存中分配空间；

(3) 把程序集中的可执行代码发送给实时（Just-in-Time, JIT）编译器，把其中的一部分编译成本机代码。

程序集中的可执行代码在需要的时候由JIT编译器编译，然后它就被缓存以备在后来的程序中执行。使用这个方法意味着不被调用的代码不会被编译成本机代码，而且被调用到的代码只被编译一次。

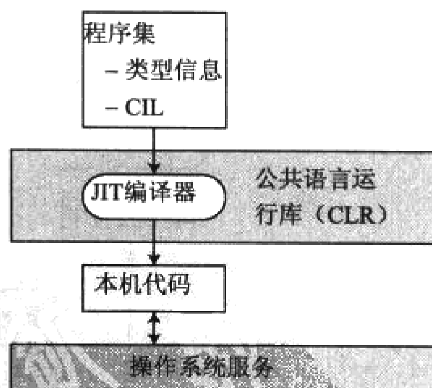



图1-4 运行时编译成本机代码

一旦CIL被编译成本机代码，CLR就在它运行时管理它，执行像释放无主内存、检查数组边界、检查参数类型和管理异常之类的任务。这产生两个重要的术语。

- **托管代码**：为.NET框架编写的代码称为托管代码（managed code），需要CLR。
- **非托管代码**：不在CLR控制之下运行的代码，比如Win32 C/C++ DLL，称为非托管代码。

微软公司还提供了一个称为本机映像生成器或*Ngen*的工具，可以把一个程序集转换成当前处理器的本机代码。经过Ngen处理过的代码免除了运行时的JIT编译过程。

## 编译和执行综述

无论原始源文件的语言是什么，都遵循同样的编译和执行过程。图1-5阐述了三个用不同语言编写的程序完整的编译和运行期过程。

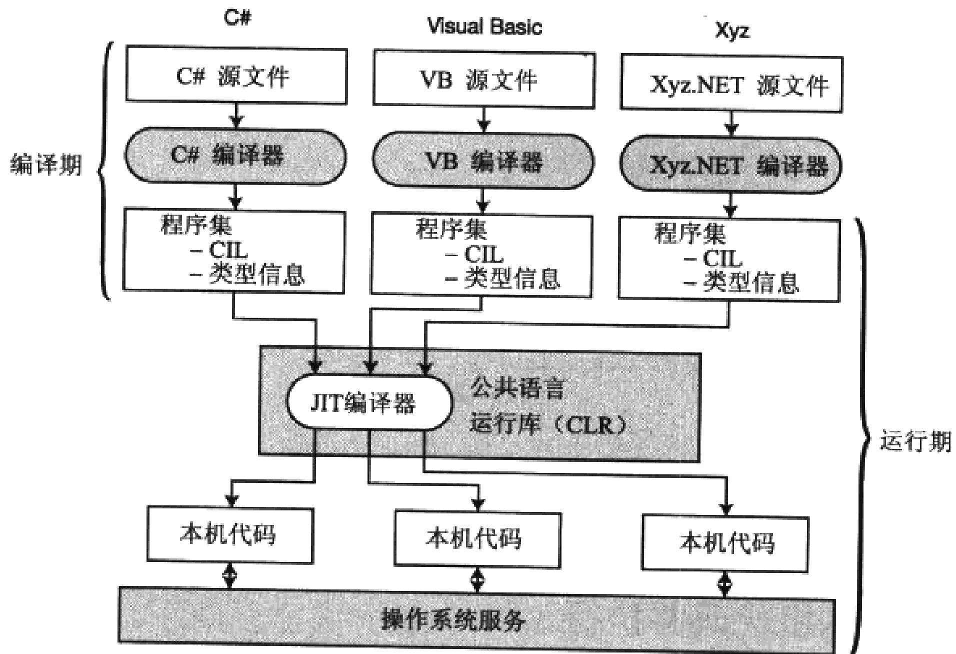



图1-5　编译期和运行期过程综述

## 1.5　CLR

.NET框架的核心组件是CLR，它在操作系统的顶层并管理程序的执行，如图1-6所示。CLR

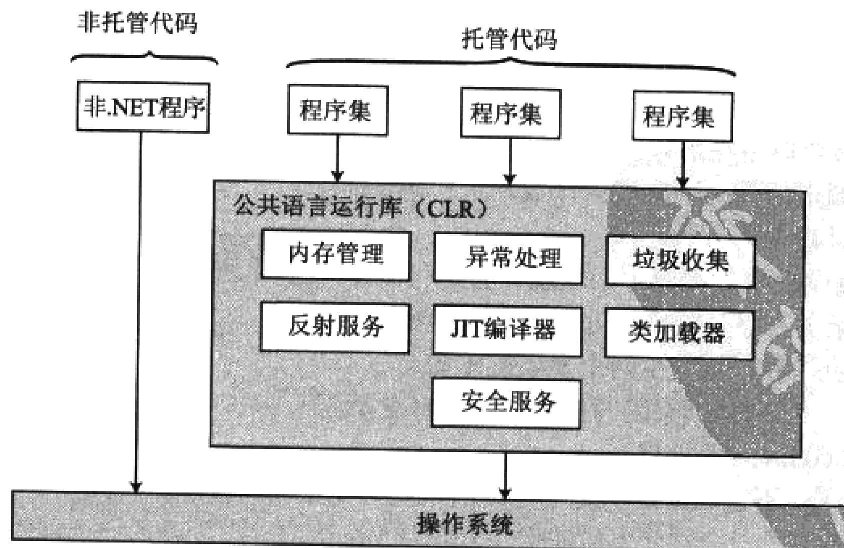



图1-6　CLR概述

还提供下列服务：

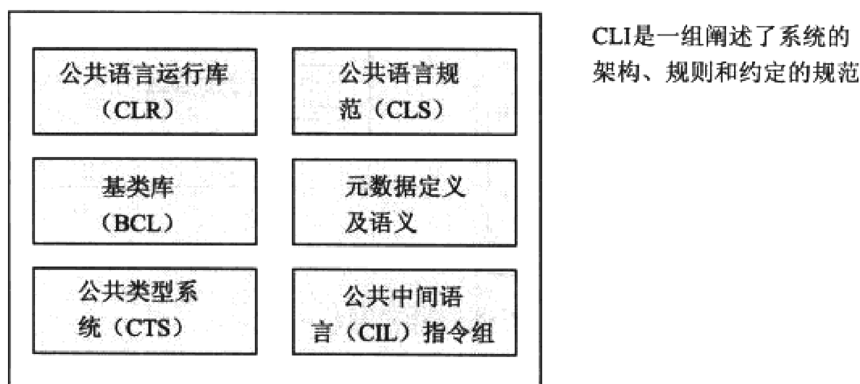
- 自动垃圾收集；
- 安全和认证；
- 通过访问BCL得到广泛的编程功能，包括如Web服务和数据服务之类的功能。

## 1.6 CLI

每种编程语言都有一组内置的类型，用来表示如整数、浮点数和字符等之类的对象。在历史上，这些类型的特征在编程语言之间和平台之间都不同。例如，组成整数的位数在不同的语言和平台之间就有很大差别。

然而，这种统一性的缺乏使我们难以让程序和其他使用不同语言编写的程序及库一起良好协作。为了有序和协作，必须有一组标准。

CLI（Common Language Infrastructure，公共语言基础结构）就是这样一组标准，它把所有.NET框架的组件连结成一个内聚的、一致的系统。它展示了系统的概念和架构，并详细说明了所有软件都必须坚持的规则和约定。CLI的组成如图1-7所示。




CLI是一组阐述了系统的架构、规则和约定的规范

图1-7 CLI的组成

CLI和C#都已经被Ecma International批准为开放的国际标准规范。（“Ecma”本来是Europen Computer Manufacturers Association［欧洲计算机制造商协会］的缩写，但现在就是一个词。）Ecma的成员包括微软、IBM、惠普、Adobe以及许多其他和计算机及消费电子有关的公司。

### CLI 的重要组成部分

虽然大多数程序员不需要了解CLI规范的细节，但至少应该熟悉公共类型系统和公共语言规范的含意和目的。

#### 1. 公共类型系统

CTS（Common Type System，公共类型系统）定义了那些在托管代码中一定会使用的类型的特征。CTS的一些重要方面如下：

- CTS定义了一组丰富的内置类型，以及每种类型确定的、详细的特性。

- .NET兼容编程语言提供的类型通常映射到CTS中已定义的内置类型集的某一个特殊子集。
- CTS最重要的特征之一是所有类型都继承自公共的基类——`object`。

**2. 公共语言规范**

CLS（Common Language Specification，公共语言规范）详细说明了一个.NET兼容编程语言的规则、属性和行为。

其主题包括数据类型、类结构和参数传递。

## 1.7 缩写回顾

本章包含了许多.NET缩写，所以最后再加入一个图1-8帮助你直观地理解它们。

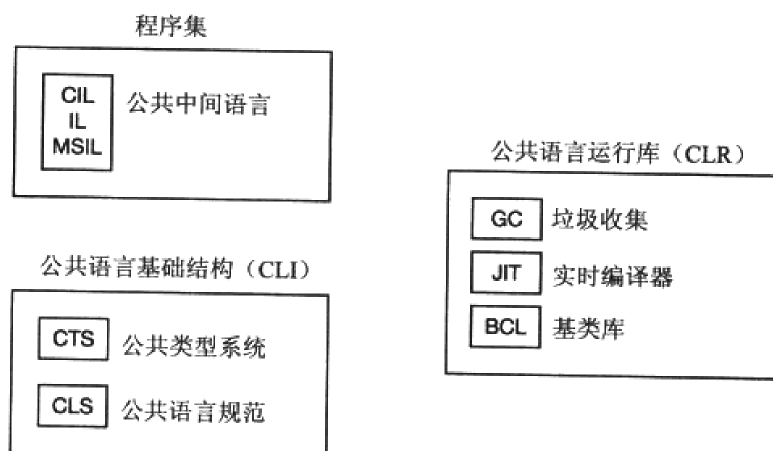



图1-8　.NET缩写

# 第 2 章 C#编程概述

**本章内容**

- 一个简单的C#程序
- 标识符和关键字
- Main：程序的起始点
- 空白
- 语句
- 从程序中输出文本
- 注释

## 2.1 一个简单的 C#程序

本章将为学习C#打基础。因为本书中会广泛地使用代码示例，所以我们先来看看C#程序的样子，还有它的不同部分代表什么意思。

我们从一个简单程序开始，逐个解释它的各组成部分。这里将会介绍许多主题，从C#程序的结构到产生程序屏幕输出的方法。

有这些源代码作为初步铺垫，我就可以在余下的文字中自由地使用代码示例了。因此，与以后的章节中详细阐述一两个主题不同，本章将接触到很多主题并只有最少的解释。

让我们从观察一个简单的C#程序开始。完整的源程序在图2-1上面的阴影区域中。如图所示，代码包含在一个名称为SimpleProgram.cs的文本文件里。当你阅读它时，不要担心能否理解所有的细节。表2-1对代码进行了逐行描述。

表2-1 SimpleProgram程序的逐行描述

| 行 号 | 描 述 |
|---|---|
| 行1 | 告诉编译器这个程序使用System命名空间的类型 |
| 行3 | 声明一个新命名空间，名称为Simple<br>• 新命名空间从第4行的左大括号开始一直延伸到第12行与之对应的右大括号<br>• 在这部分里声明的任何类型都是该命名空间的成员 |
| 行5 | 声明一个新的类类型，名称为Program<br>• 任何在第6行和第7行的两个大括号中间声明的成员都是组成这个类的成员 |

（续）

| 行　　号 | 描　　述 |
| --- | --- |
| Line 7 | 声明一个名称为Main的方法作为类Program的成员<br>• 在这个程序中，Main是Program类的唯一成员<br>• Main是一个特殊函数，编译器用它作为程序的起始点 |
| Line 9 | 只包含一条单独的、简单的语句，这一行组成了Main的方法体<br>• 简单语句以一个分号结束<br>• 这条语句使用命名空间System中的一个名称为Console的类打印出消息到屏幕窗口<br>• 没有第1行的using语句，编译器就不会知道在哪里寻找类Console |

- 当代码被编译执行时，它在一个屏幕窗口中显示字符串“Hi there!”。
- 有一行包含两个相邻的斜杠。这两个字符以及这一行中它们之后的所有内容都会被编译器忽略。这叫做单行注释。

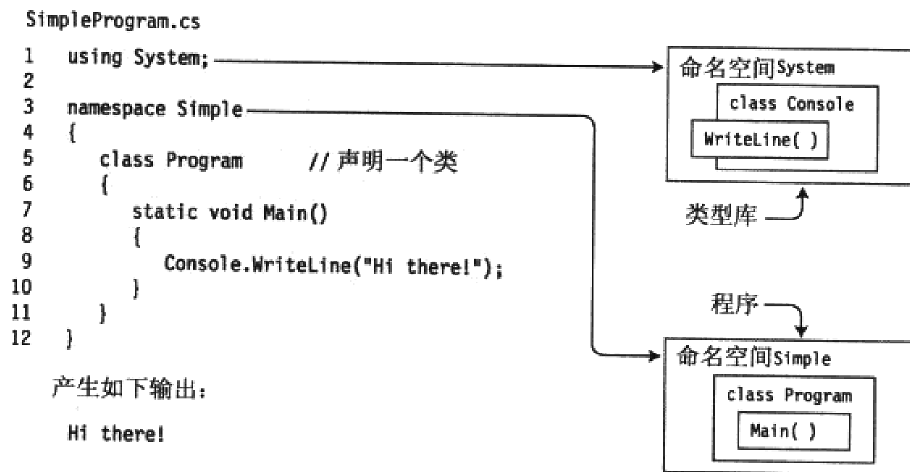



图2-1　SimpleProgram程序的逐行描述

## SimpleProgram 的补充说明

C#程序由一个或多个类型声明组成。本书的大部分内容都是用来解释可以在程序中创建和使用的不同类型。程序中的类型可以以任何顺序声明。在SimpleProgram中，只声明了class类型。

命名空间是和一个名称相关联的一组类型声明。SimpleProgram使用两个命名空间。它创建了一个名称为Simple的新命名空间，并使用了一个名称为System的预定义命名空间。

要编译这个程序，可以使用Visual Studio或命令行编译器。如果使用命令行编译器，最简单的形式是使用下面的命令：

```
csc SimpleProgram.cs
```

在这条命令中，csc是命令行编译器的名称，SimpleProgram.cs是源文件的名称。

## 2.2　标识符和关键字

标识符是一种字符串，用来命名如变量、方法、参数和许多后面将要阐述的其他程序结构之

类的东西。

可以通过把有意义的词连接成一个单独的描述性名称来创建自文档化（self-documenting）的标识符，可以使用大写和小写字母（如CardDeck、PlayersHand、FirstName和SocSecurityNum）。在标识符中某些特定的位置允许或不允许特定的字符，这些规则如图2-2所示。

- 字母和下划线（a-z、A-Z和_）可以用在任何位置。
- 数字不能放在首位，但可以在其他的任何地方。
- @字符可以放在标识符的首位，但不能放在其他任何地方。虽然允许使用@字符，但不推荐作为常用字符。

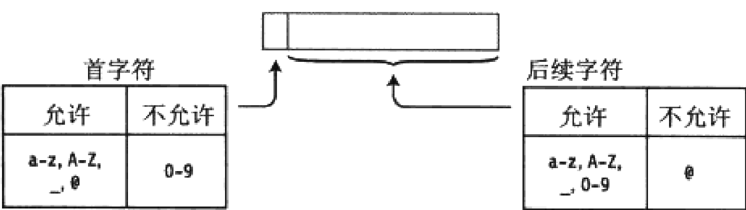



图2-2 标识符中允许的字符

标识符区分大小写。例如，变量名myVar和MyVar是不同的标识符。但是，不应该使用只有字母的大小写不同的多个标识符。

举个例子，在下面的代码片段中，变量的声明都是有效的，并声明了不同的整型变量。但使用如此相似的名称会使代码更易出错并更难调试，以后代码调试工作将不会轻松。

```
// 语法上有效，但是不应这样做
int totalCycleCount;
int TotalCycleCount;
int TotalcycleCount;
```

## 2.2.1 命名约定

C#语言规范建议使用特定大小写约定创建标识符。表2-2总结并描述了建议使用的大小写约定。

对于大多数标识符，应该使用Pascal大小写风格。在这种风格中，连接成标识符的每个单词都首字母大写——例如FirstName和LastName。

**表2-2 推荐的标识符命名风格**

| 风格名称 | 描 述 | 使用建议 | 示 例 |
|---|---|---|---|
| Pascal大小写 | 标识符中每个单词都首字母大写 | 用于类型名和成员名 | CarDeck，DealersHand |
| Camel大小写 | 除第一个单词以外，标识符中所有的单词都首字母大写 | 用于本地变量和方法参数 | totalCycleCount，randomSeedParam |
| 全大写 | 标识符由全大写字母组成 | 仅用于缩写词 | IO，DMA，XML |

虽然这是建议的指导方针，但很多组织使用其他的约定，尤其是在成员字段的命名方面，这些内容将在下一章介绍。有两个公共约定如下所示。

- 字段名称使用下划线开头：_HighTemp、_LowTemp。
- 字段名称使用m_开头：m_HighTemp、m_LowTemp。

这两种方法都有优势，能立刻显示这些标识符是字段名称。这些形式还能让Visual Studio的Intelli Sense（智能感知）特性在弹出窗口中把所有的字段组合在一起。

### 2.2.2 关键字

关键字是用来定义C#语言的字符串记号。表2-3列出了完整的C#关键字表。

关于关键字，一些应该知道的重要内容如下：

- 关键字不能被用作变量名或任何其他形式的标识符，除非以@字符开始。
- 所有C#关键字全部都由小写字母组成，但是.NET类型名使用Pascal大小写约定。

表2-3 C#关键字

| | | | | | | |
|---|---|---|---|---|---|---|
| abstract | const | extern | int | out | short | typeof |
| as | continue | false | interface | override | sizeof | uint |
| base | decimal | finally | internal | params | stackalloc | ulong |
| bool | default | fixed | is | private | static | unchecked |
| break | delegate | float | lock | protected | string | unsafe |
| byte | do | for | long | public | struct | ushort |
| case | double | foreach | namespace | readonly | switch | using |
| catch | else | goto | new | ref | this | virtual |
| char | enum | if | null | return | throw | void |
| checked | event | implicit | object | sbyte | true | volatile |
| class | explicit | in | operator | sealed | try | while |

上下文关键字是仅在特定的语言结构中充当关键字的标识符。在那些位置，它们有特别的含意。但和关键字不同，关键字不能被用作标识符，而上下文关键字可以在其他部分代码中被用作标识符。上下文关键字如表2-4所示。

表2-4 C#的上下文关键字

| | | | | | | |
|---|---|---|---|---|---|---|
| ascending | by | descending | equals | from | get | group |
| into | join | let | on | orderby | partial | select |
| set | value | where | yield | | | |

## 2.3 Main：程序的起始点

每个C#程序必须有一个类带有Main方法（函数）。在先前所示的SimpleProgram程序中，它被声明在Program类中。

- 每个C#程序的可执行起始点在Main中的第一条指令。
- Main必须首字母大写。
- Main的最简单形式如下：

```
static void Main( )
{
  更多语句
}
```

## 2.4 空白

程序中的空白指的是没有可视化输出的字符。源代码中的空白将被编译器忽略，但程序员用它们使代码更清晰易读。空白字符包括：

- 空格（Space）
- 制表符（Tab）
- 换行符
- 回车符

例如，下面的代码段会被编译器完全相同地对待而不管它们表面上的区别。

```
// 很好的格式
Main()
{
    Console.WriteLine("Hi, there!");
}

// 连在一起
Main(){Console.WriteLine("Hi, there!");}
```

## 2.5 语句

C#的语句和C、C++的语句非常相似。本节将介绍语句的常用形式，详细的语句形式将在第9章介绍。

### 2.5.1 简单语句

语句是描述一个类型或告诉程序去执行一个动作的一条源代码指令。

- 简单语句以一个分号结束。

例如，下面的代码是一个由两条简单语句组成的序列。第一条语句定义了一个名称为var1的变量，并初始化它的值为5。第二条语句打印变量var1的值到屏幕窗口。

```
int var1 = 5;
System.Console.WriteLine("The value of var1 is {0}", var1);
```

### 2.5.2 块

块是一个由成对大括号包围的0条或多条语句序列，它在语法上相当于一条语句。

可以使用之前示例中的两条语句创建一个块。用大括号把语句包围起来，如下面的代码所示：

```
{
    int var1 = 5;
    System.Console.WriteLine("The value of var1 is {0}", var1);
}
```

关于块，一些应该知道的重要内容如下：

- 可以在任何语法上只需要一条语句而你需要的行为又要求一条以上的简单语句的情况下使用块。
- 有些特定的程序结构只能使用块。在这些结构中，不能用简单语句替代块。
- 简单语句以分号结束，但块后面不跟分号。（虽然编译器允许这样，但这不是好的风格。）

```
{           以分号结束
                ↓                                              以分号结束
    int var2 = 5;                                                  ↓
    System.Console.WriteLine("The value of var1 is {0}", var1);
}
 ↑ 不以分号结束
```

## 2.6　从程序中输出文本

控制台窗口是一种简单的命令提示窗口，允许程序显示文本并从键盘接受输入。BCL提供一个名称为Console的类（在System命名空间中），该类包含了输入和输出数据到控制台窗口的方法。

### 2.6.1　Write

Write是Console类的成员，它把一个文本字符串发送到程序的控制台窗口。最简单的情况下，Write将文本字符串发送到窗口，字符串必须使用双引号括起来。

下面这行代码展示了一个使用Write成员的示例：

```
Console.Write("This is trivial text.");
                      ↑
                   输出字符串
```

这段代码在控制台窗口产生如下输出：

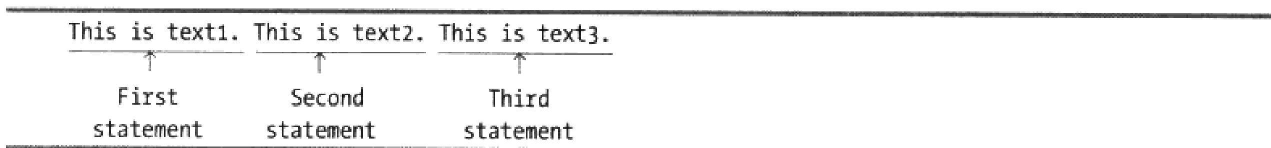
```
This is trivial text.
```

另外一个示例是下面的代码，发送了三个文本字符串到程序的控制台窗口：

```
System.Console.Write ("This is text1.");
System.Console.Write ("This is text2.");
System.Console.Write ("This is text3.");
```

这段代码产生的输出如下，注意，Write没有在字符串后面添加换行符，所以三条语句都输

出到同一行。




## 2.6.2　WriteLine

WriteLine是Console的另外一个成员，它和Write实现相同的功能，但会在每个输出字符串的结尾添加一个换行符。

例如，如果使用先前的代码，用WriteLine替换掉Write，输出就会分隔在多行：

```
System.Console.WriteLine("This is text 1.");
System.Console.WriteLine("This is text 2.");
System.Console.WriteLine("This is text 3.");
```

这段代码在控制台窗口产生如下输出：

```
This is text 1.
This is text 2.
This is text 3.
```

## 2.6.3　格式字符串

Write语句和WriteLine语句的常规形式中可以有一个以上的参数。

- 如果不只一个参数，参数间用逗号分隔。
- 第一个参数必须总是字符串，称为格式字符串。
- 格式字符串可以包含替代标记。替代标记在格式字符串中标出位置，在输出串中该位置将用一个值来替代。它由一个整数及括住它的一对大括号组成，其中整数就是替换值的数字位置。
- 紧跟着格式字符串的参数称为替换值，这些替换值从0开始编号。

语法如下：

```
Console.WriteLine(格式字符串（含替代标记），替换值0，替换值1，替换值2，…);
```

例如，下面的语句有两个替代标记，编号0和1；以及两个替换值，它们的值分别是3和6。

```
                                        替代标记
                                        ↓        ↓
Console.WriteLine("Two sample integers are {0} and {1}.", 3, 6);
                             ↑                             ↑
                          格式字符串                      替换值
```

这段代码在屏幕上产生如下输出：

```
Two sample integers are 3 and 6.
```

### 2.6.4　多重标记和值

在C#中，可以使用任意数量的替代标记和任意数量的值。

- 值可以以任何顺序使用。
- 值可以在格式字符串中替换任意次。

例如，下面的语句使用了三个标记但只有两个值。请注意，值1被用在了值0之前，而且值1被使用了两次。

```
Console.WriteLine("Three integers are {1}, {0} and {1}.", 3, 6);
```

这段代码在屏幕上显示如下：

```
Three integers are 6, 3 and 6.
```

标记不能试图引用替换值列表以外位置的值。如果这样做了，不会产生编译错误，但会产生运行时错误（称为异常）。

例如，在下面的语句中有两个替换值，在位置0和1。而第二个标记引用了位置2，位置2并不存在。这将会产生一个运行时错误。

```
                                                位置 0  位置 1
                                                     ↓    ↓
Console.WriteLine("Two integers are {0} and {2}.", 3, 6);    // Error!
                                             ↑
                                       位置2的值不存在
```

## 2.7　注释

你已经见过单行注释了，所以这里将讨论第二种行内注释——带分隔符注释，并提及第三种称为文档注释的类型。

- 带分隔符注释有开始标记和结束标记。
- 标记对之间的文本会被编译器忽略。
- 带分隔符注释可以跨任意多行。

例如，下面的代码展示了一个跨多行的带分隔符注释。

```
↓ 跨多行注释的开始
/*
   这段文本将被编译器忽略
   带分隔符的注释与单行注释不同
   带分隔符的注释可以跨越多行
*/
↑ 注释结束
```

带分隔符注释还可以只包括行的一部分。例如，下面的语句展示了行中间注释出的文本。该

声明的结果只有一个单独变量var2。

```
  注释开始
     ↓
int /*var 1,*/ var2;
              ↑
           注释结束
```

**说明** 在C#中，单行注释和带分隔符注释的行为与在C和C++中的相同。

## 2.7.1 关于注释的补充

关于注释，有几点其他的重要内容需要知道。

- 嵌套注释是不允许的，同一时间只能有一个注释起作用。
- 首先开始的注释直到它的范围结束都有效。注释类型的范围如下：
  - 对于单行注释，一直到行结束。
  - 对于带分隔符注释，直至遇到第一个结束分隔符。

下面的注释方式是不正确的：

```
↓ 创建注释
/* 尝试嵌套注释
   /* ← 它将被忽略，因为它在一个注释的内部
      内部注释
   */ ← 注释结束，因为它是遇到的第一个结束分隔符
*/  ← 产生语法错误，因为没有开始分隔符

↓ 创建注释                  ↓ 它将被忽略，因为它在一个注释的内部
// 单行注释                 /* 嵌套注释？
                        */  ← 产生语法错误，因为没有开始分隔符
```

## 2.7.2 文档注释

C#还提供第三种注释类型：文档注释。文档注释包含XML文本，可以用于产生程序文档。这种类型的注释看起来像单行注释，但它们有三个斜杠而不是两个。文档注释会在第25章阐述。

下面的代码展示了文档注释的形式：

```
/// <summary>
/// This class does...
/// </summary>
class Program
{
   ...
```

### 2.7.3　注释类型总结

行内注释是被编译器忽略但被包含在代码中以说明代码的文本片段。程序员在他们的代码中插入注释以解释和文档化代码。表2-5总结了注释的类型。

表2-5　注释类型

| 类　型 | 开始 | 结束 | 描　述 |
|---|---|---|---|
| 单行注释 | // | | 从开始标记到该行行尾的文本被编译器忽略 |
| 带分隔符注释 | /* | */ | 从开始标记到结束标记之间的文本被编译器忽略 |
| 文档注释 | /// | | 这种类型的注释包含XML文本，可以使用工具生成程序文档 |

第 3 章

# 类型、存储和变量

3

**本章内容**

- ❑ C#程序是一组类型声明
- ❑ 类型是一种模板
- ❑ 实例化类型
- ❑ 数据成员和函数成员
- ❑ 预定义类型
- ❑ 用户定义类型
- ❑ 栈和堆
- ❑ 值类型和引用类型
- ❑ 变量

## 3.1 C#程序是一组类型声明

如果广泛地描述C和C++程序源代码的特征，可以说C程序是一组函数和数据类型，C++程序是一组函数和类，然而C#程序是一组类型声明。

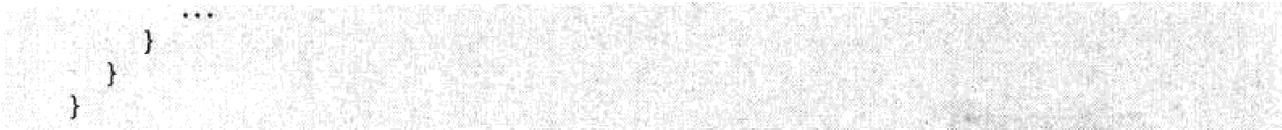
- ❑ C#程序或DLL的源代码是一组类型声明。
- ❑ 对于可执行程序，类型声明中必须有一个包含Main方法的类。
- ❑ 命名空间是一种把相关的类型声明分组并命名的方法。既然程序是一组相关的类型声明，那么通常会把程序声明在创建的命名空间内部。

例如，下面是一个由三个类型声明组成的程序。这三个类型被声明在一个名称为MyProgram的新命名空间内部。

```
namespace MyProgram                          // 创建新的命名空间
{
   DeclarationOfTypeA                        // 声明类型

   DeclarationOfTypeB                        // 声明类型

   class C                                   // 声明类型
   {
      static void Main()
      {
```



```
            ...
        }
    }
}
```

命名空间将在第10章详细阐述。

## 3.2　类型是一种模板

既然C#程序就是一组类型声明，那么学习C#就是学习如何创建和使用类型。所以，需要做的第一件事情就是了解什么是类型。

可以把类型想象成一个用来创建数据结构的模板。模板本身并不是数据结构，但它详细说明了由该模板构造的对象的特征。

类型由下面的元素定义：

- 名称
- 用于保存数据成员的数据结构
- 一些行为及约束条件

例如，图3-1阐明了short类型和int类型的组成元素。

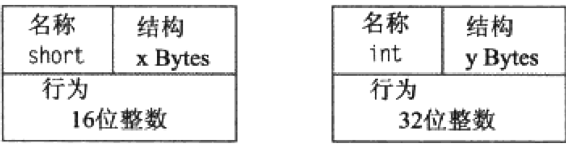



图3-1　类型是一种模板

## 3.3　实例化类型

从某个类型模板创建实际的对象，称为实例化该类型。

- 通过实例化类型而创建的对象被称为类型的对象或类型的实例。这两个术语可以互换。
- 在C#程序中，每个数据项都是某种类型的实例。这些类型可以是语言自带的，可以是BCL或其他库提供的，也可以是程序员定义的。

图3-2阐明了两种预定义类型对象的实例化。

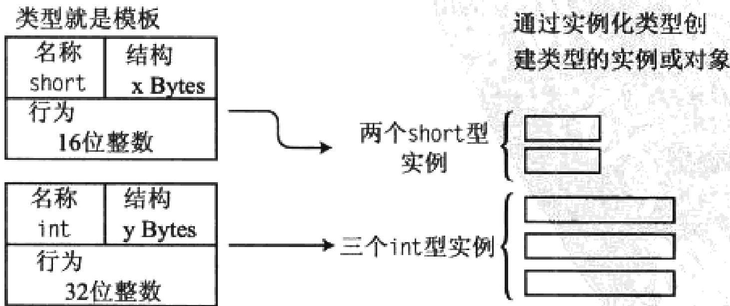



图3-2　通过实例化类型创建实例

## 3.4 数据成员和函数成员

short、int和long等类型称为简单类型。这种类型只能存储一个数据项。其他的类型可以存储多个数据项。比如数组（array）类型就可以存储多个同类型的数据项。这些数据项称为数组元素。可以通过数字来引用这些元素，这些数字称为索引。数组将会在第14章详述。

### 成员的类别

然而另外一些类型可以包含许多不同类型的数据项。这些类型中的数据项个体称为成员，并且与数组中使用数字来引用成员不同，这些成员有独特的名称。

有两种成员：数据成员和函数成员。

- 数据成员保存了与这个类的对象或类本身相关的数据。
- 函数成员执行代码。函数成员定义类型的行为。

例如，图3-3列出了类型XYZ的一些数据成员和函数成员。它包含两个数据成员和两个函数成员。

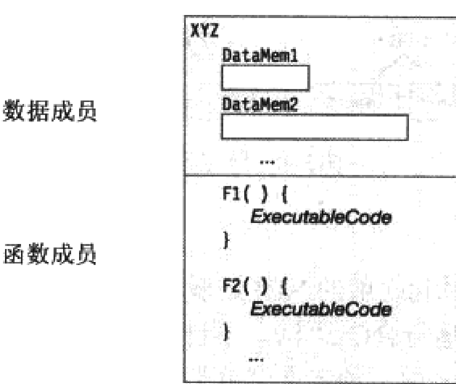



图3-3 类型包含数据成员和函数成员

## 3.5 预定义类型

C#提供了15种预定义类型，如图3-4所示。它们列在表3-1和表3-2中，其中包括13种简单类型和2种非简单类型。

所有预定义类型的名称都由全小写的字母组成。预定义的简单类型包括：

- 11种数值类型。
  - 不同长度的有符号和无符号整数类型。
  - 浮点数类型float和double。
  - 一种称为decimal的高精度小数类型。与float和double不同，decimal类型可以准确地表示分数。decimal类型常用于货币的计算。
- 一种Unicode字符类型char。
- 一种布尔类型bool。bool类型表示布尔值并且必须为true或false。

**说明** 与C和C++不同，在C#中的数值类型不具有布尔意义。

两种非简单类型是：

- string，它是一个Unicode字符数组。
- object，它是所有其他类型的基类。

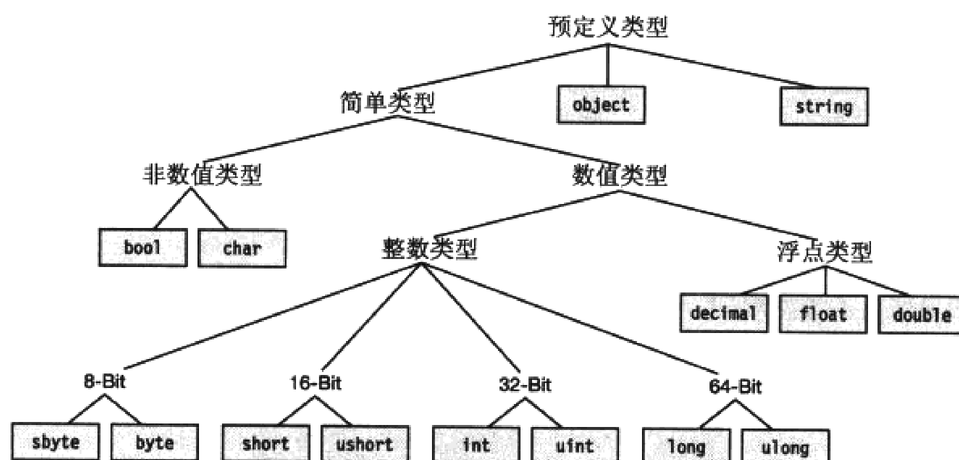



图3-4 预定义类型

## 预定义类型的补充

所有预定义类型都直接映射到底层的.NET类型。C#的类型名称就是.NET类型的别名，所以使用.NET的类型名称也能很好地符合C#语法，不过并不鼓励这样做。在C#程序中，应该尽量使用C#类型名称而不是.NET类型名称。预定义简单类型表示一个单一的数据项。表3-1列出了这些类型，并同时列出了它们的取值范围和对应的底层.NET类型。

**表3-1 预定义简单类型**

| 名 称 | 含 义 | 范 围 | .NET框架类型 | 默认值 |
|---|---|---|---|---|
| sbyte | 8位无符号整数 | -128～127 | System.SByte | 0 |
| byte | 8位无符号整数 | 0～255 | System.Byte | 0 |
| short | 16位无符号整数 | -32 768～32 767 | System.Int16 | 0 |
| ushort | 16位无符号整数 | 0～65 535 | System.UInt16 | 0 |
| int | 32位有符号整数 | -2 147 483 648～2 147 483 647 | System.Int32 | 0 |
| uint | 32位无符号整数 | 0–4 294 967 295 | System.UInt32 | 0 |
| long | 64位有符号整数 | -9 223 372 036 854 775 808 ～9 223 372 036 854 775 807 | System.Int64 | 0 |
| ulong | 64位无符号整数 | 0～18 446 744 073 709 551 615 | System.UInt64 | 0 |
| float | 单精度浮点数 | $1.5\times10^{-45}$～$3.4\times10^{38}$ | System.Single | 0.0f |
| double | 双精度浮点数 | $5\times10^{-324}$～$1.7\times10^{308}$ | System.Double | 0.0d |
| bool | 布尔型 | true false | System.Boolean | false |
| char | Unicode 字符串 | U+0000～U+ffff | System.Char | \x0000 |
| decimal | 小数类型的有效数字精度为28位 | $\pm1.0\times10^{28}$～$\pm7.9\times10^{28}$ | System.Decimal | 0m |

非简单预定义类型稍微复杂一些。类型string的值可以包含0个或多个Unicode字符。object类型是系统中所有其他类型的基类，包括预定义简单类型，如表3-2所示。

表3-2 预定义非简单类型

| 名称 | 含义 | .NET框架类型 |
|---|---|---|
| object | 所有其他类型的基类 | System.Object |
| string | Unicode字符序列 | System.String |

## 3.6 用户定义类型

除了C#提供的15种预定义类型，还可以创建自己的用户定义类型。有6种类型可以由用户自己创建，它们是：

- 类类型（class）
- 结构类型（struct）
- 数组类型（array）
- 枚举类型（enum）
- 委托类型（delegate）
- 接口类型（interface）

类型通过*类型声明*创建，类型声明包含以下信息：

- 要创建的类型的种类；
- 新类型的名称；
- 对类型中每个成员的声明（名称和规格）。array和delegate类型除外，它们不含有命名成员。

一旦声明了类型，就可以创建和使用这种类型的对象，就像它们是预定义类型一样。图3-5概括了预定义类型和用户定义类型的使用。使用预定义类型是一个单步过程，简单地实例化对象即可。使用用户定义类型是一个两步过程：先声明类型，然后实例化该类型的对象。

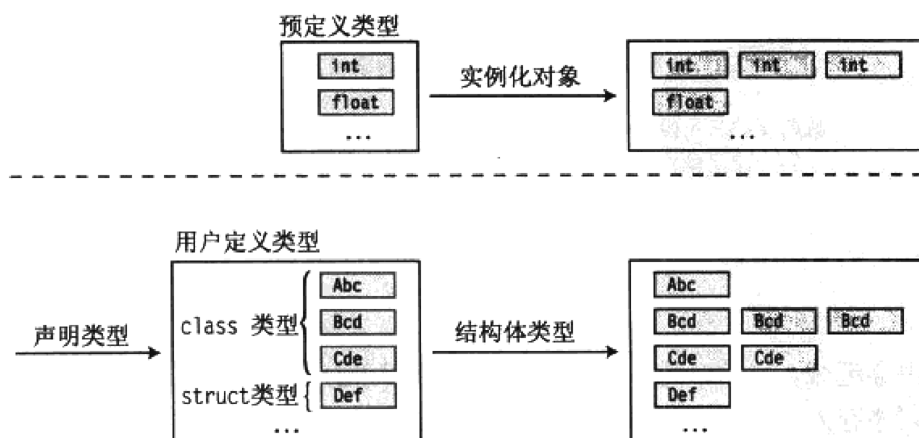



图3-5 预定义类型只需要进行实例化；用户定义类型需要两步：声明和实例化

## 3.7　栈和堆

程序运行时，它的数据必须存储在内存中。一个数据项需要多大的内存、存储在什么地方、以及如何存储都依赖于该数据项的类型。

运行中的程序使用两个内存区域来存储数据：栈和堆。

### 3.7.1　栈

系统接管所有的栈操作。作为程序员，不需要显式地对它做任何事情。但了解栈的基本功能可以更好地了解程序在运行时正在做什么，并能更好地了解C#文档和图书。

栈是一个内存数组，是一个LIFO（last-in first-out，后进先出）的数据结构。栈存储几种类型的数据：

- 某些类型变量的值；
- 程序当前的执行环境；
- 传递给方法的参数。

**栈的特征**

栈有如下几个普遍特征（见图3-6）：

- 数据只能从栈的顶端插入和删除；
- 把数据放到栈顶称为入栈（push）；
- 从栈顶删除数据称为出栈（pop）。

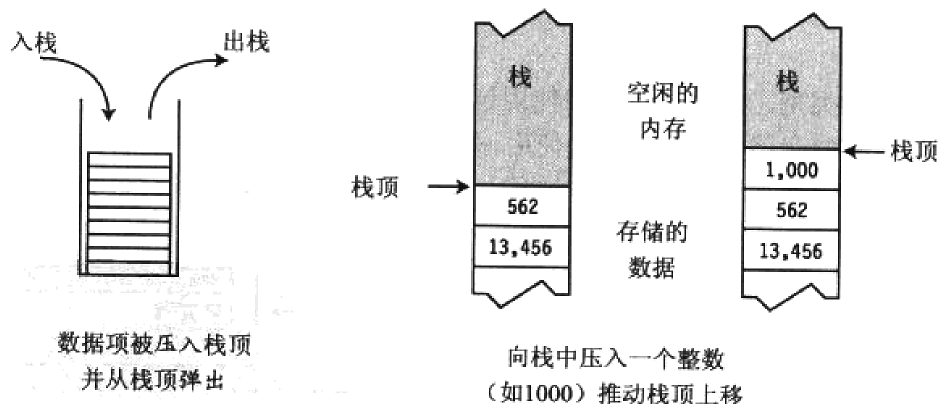



图3-6　入栈和出栈

### 3.7.2　堆

堆是一块内存区域，在堆里可以分配大块的内存用于存储某类型的数据。与栈不同，堆里的内存可以任意顺序存入和移除。图3-7展示了一个在堆里放了4项数据的程序。

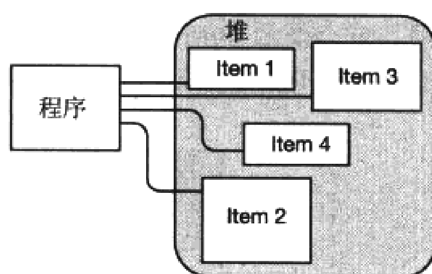



图3-7 内存堆

虽然程序可以在堆里保存数据，但并不能显式地删除它们。CLR的自动GC（Garbage Collector，垃圾收集器）在判断出程序的代码将不会再访问某数据项时，自动清除无主的堆对象。我们因此可以不再操心这项使用其他编程语言时非常容易出错的工作了。图3-8阐明了垃圾收集过程。

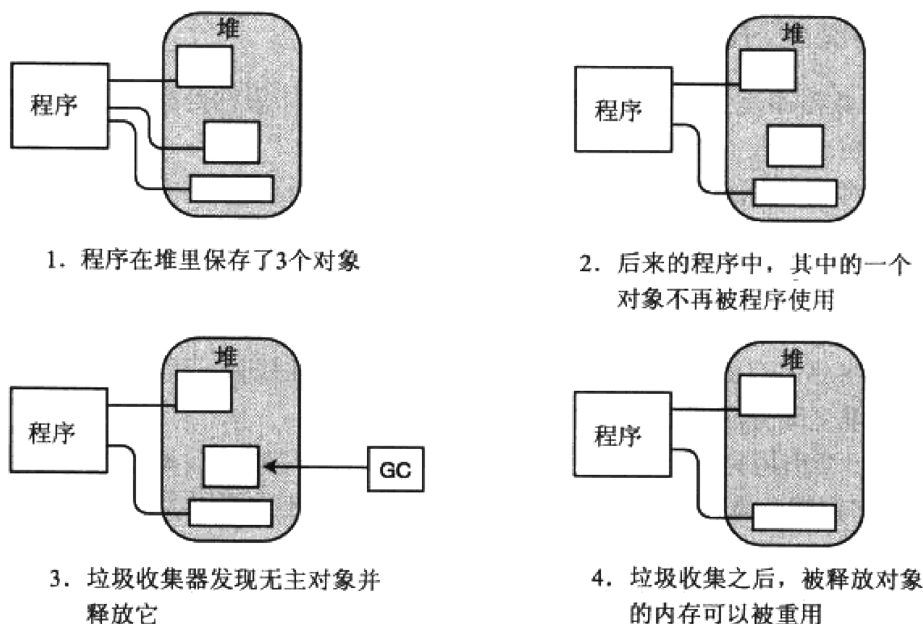



图3-8 堆中的自动垃圾收集

## 3.8 值类型和引用类型

数据项的类型定义了存储数据需要的内存大小、组成该类型的数据成员以及该类型能执行的函数。类型还决定了对象在内存中的存储位置——栈或堆。

类型被分为两种：值类型和引用类型，这两种类型的对象在内存中的存储方式不同。

- 值类型只需要一段单独的内存，用于存储实际的数据。
- 引用类型需要两段内存：
  - 第一段存储实际的数据，它总是位于堆中。
  - 第二段是一个引用，指向数据在堆中的存放位置。

数据如果不是其他类型的成员，就会像图3-9所示那样存储。对于值类型，数据存放在栈里。对于引用类型，实际数据存放在堆里而引用存放在栈里。

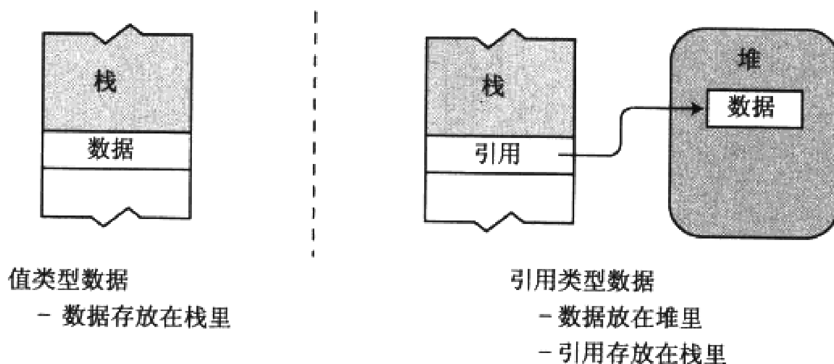



图3-9　非成员数据的存储

### 3.8.1　存储引用类型对象的成员

图3-9阐明了当数据不是另一个类型的成员时如何存储。如果它是另一个类型的成员，那么它的存储会有些不同。

- 引用类型对象的数据部分始终存放在堆里，如图3-9所示。
- 值类型对象，或引用类型数据的引用部分可以存放在堆里，也可以存放在栈里，这依赖于实际环境。

例如，假设有一个引用类型的实例，名称为MyType，它有两个成员：一个值类型成员和一个引用类型成员。它将如何存储呢？是否是值类型的成员存储在栈里，而引用类型的成员如图3-9所示的那样在栈和堆之间分成两半呢？答案是否定的。

请记住，对于一个引用类型，其实例的数据部分始终存放在堆里。既然两个成员都是对象数据的一部分，那么它们都会被存放在堆里，无论它们是值类型还是引用类型。图3-10阐明了MyType的情形：

- 尽管成员A是值类型，但它也是MyType实例数据的一部分，因此和对象的数据一起被存放在堆里。
- 成员B是引用类型，所以它的数据部分会始终存放在堆里，正如图中“数据”框所示。不同的是，它的引用部分也被存放在堆里，封装在MyType对象的数据部分中。

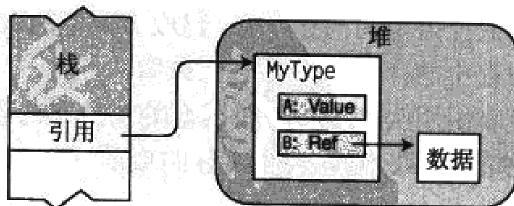



图3-10　引用类型成员数据的存储

**说明** 对于引用类型的任何对象，它所有的数据成员都存放在堆里，无论它们是值类型还是引用类型。

### 3.8.2 C#类型的分类

表3-3列出了C#中可以使用的所有类型以及它们的类别：值类型或引用类型。每种类型都将在后面的内容中阐述。

表3-3 C#中的值类型和引用类型

| | 值类型 | | | 引用类型 |
|---|---|---|---|---|
| 预定义类型 | sbyte | byte | float | object |
| | short | ushort | double | string |
| | int | uint | char | |
| | long | ulong | decimal | |
| | bool | | | |
| 用户定义类型 | struct | | | class |
| | enum | | | interface |
| | | | | delegate |
| | | | | array |

## 3.9 变量

一种多用途的编程语言必须允许程序存取数据，而这正是通过变量实现的。

- 变量是一个名称，表示程序执行时存储在内存中的数据。
- C#提供了4种变量，每一种都将详细讨论。表3-4列出了变量的种类：

表3-4 4种变量

| 名 称 | 类型的成员 | 描 述 |
|---|---|---|
| 本地变量 | 否 | 在方法的作用域保存临时数据 |
| 字段 | 是 | 保存和类型相关的数据 |
| 参数 | 否 | 用于从一个方法到另一个方法传递数据的临时变量 |
| 数组元素 | 是 | 用于保存临时的或类型相关的数据 |

### 3.9.1 变量声明

变量在使用之前必须声明。变量声明定义了变量，并完成两件事：

- 给变量命名，并为它关联一种类型；
- 让编译器为它分配一块内存。

一个简单的变量声明至少需要一个类型和一个名称。下面的声明定义了名称为var2的变量，类型为int：

```
类型
 ↓
int var2;
     ↑
     值
```

例如，图3-11展现了4个变量的声明以及它们在栈中的位置。

```
int     var1;        // 变量类型
int     var2;        // 变量类型
float   var3;        // 变量类型
Dealer  theDealer;   // 引用类型
```

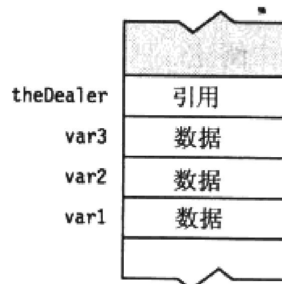



图3-11　值类型和引用类型变量的声明

### 1. 变量初始化语句

除声明变量的名称和类型以外，声明还能把它的内存初始化为一个明确的值。

**变量初始化语句**（variable）由一个等号后面跟一个初始值组成，如：

```
          初始值
            ↓
int var2 = 17;
```

无初始化语句的本地变量有一个未定义的值，在未赋值之前不能使用。试图使用未定义的本地变量会导致编译器产生一条错误信息。

图3-12在左边展示了许多本地变量声明，在右边展示了栈的构造结果。一些变量有初始化语句，其他的变量没有。

```
int     var1;               // 变量类型
int     var2 = 17;          // 变量类型
float   var3 = 26.843F;     // 变量类型
Dealer  dealer1;            // 引用类型
Dealer  dealer2 = null;     // 引用类型
```

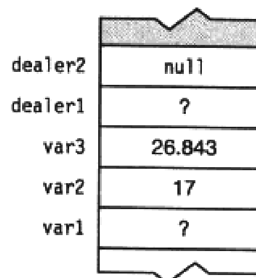



图3-12　变量初始化语句

### 2. 自动初始化

一些类型的变量如果在声明时没有初始化语句，那么会被自动设为默认值，而另一些则不能。没有自动初始化为默认值的变量在程序为它赋值之前包含未定义值。表3-5展示了哪种类型的变量会被自动初始化以及哪种类型的变量不会被初始化。我会在以后的内容中对5种变量类型进行

详细阐述。

表3-5 变量类型

| 变量类型 | 存储位置 | 自动初始化 | 用 途 |
|---|---|---|---|
| 本地变量 | 栈或者栈和堆 | 否 | 用于函数成员内部的本地计算 |
| 类字段 | 堆 | 是 | 类的成员 |
| 结构字段 | 栈或堆 | 是 | 结构的成员 |
| 参数 | 栈 | 否 | 用于把值传入或传出方法 |
| 数组元素 | 堆 | 是 | 数组的成员 |

### 3.9.2 多重变量声明

可以把多个变量声明在一条单独的声明语句中。

- 多重变量声明中的变量必须类型相同。
- 变量名必须用逗号分隔，可以在变量名后包含初始化语句。

例如，下面的代码展示了两条有效的多重变量声明语句。注意，只要使用逗号分开，初始化的变量可以和未初始化的变量混在一起。最后一条声明语句是有问题的，因为它企图在一条语句中声明两个不同类型的变量。

```
// 声明一些变量，有的被初始化，有的未被初始化
int    var3 = 7, var4, var5 = 3;
double var6, var7 = 6.52;

整型       浮点型
↓          ↓
int var8, float var9;       // 错误！多重变量声明的变量类型必须相同
```

### 3.9.3 使用变量的值

变量名代表该变量保存的值，可以通过使用变量名来使用值。

例如，var2的值被从内存获取并放在变量名的位置，像这样：

```
Console.WriteLine("{0}", var2);
```

# 第 4 章 类：基础

本章内容

- 类的概述
- 程序和类：一个快速的示例
- 声明类
- 类成员
- 创建变量和类的实例
- 为数据分配内存
- 实例成员
- 访问修饰符
- 从类的内部访问成员
- 从类的外部访问成员
- 综合应用

## 4.1 类的概述

在上一章中，我们看到C#提供了6种用户定义类型。其中最重要的是类，也是首先要阐述的。因为类在C#中是个很大的主题，关于它的讨论将会延伸到接下来的几章。

### 类是一种活动的数据结构

在面向对象的分析和设计产生之前，程序员们仅把程序当作指令的序列。那时的焦点主要放在指令的组合和优化上。随着面向对象的出现，焦点从优化指令转移到组织程序的数据和功能上来。程序的数据和功能被组织为逻辑上相关的数据项和函数的封装集合，称之为类。

类是一个能存储数据并执行代码的数据结构。它包含以下内容：

- *数据成员*，它存储与类或类的实例相关的数据。数据成员通常模拟该类所表示的现实世界事物的特性。
- *函数成员*，它执行代码。函数成员通常模拟类所表示的现实世界事物的功能和操作。

一个C#类可以有任意数目的数据成员和函数成员。成员可以是9种可能的成员类型的任意组合。这些成员类型如表4-1所示。本章所要阐述的类型——*字段*和*方法*——在表中被选中了。

表4-1 类成员的类型

| 数据成员——存储数据 | 函数成员——执行代码 | |
|---|---|---|
| ✔ 字段 | ✔ 方法 | ❑运算符 |
| ❑ 常量 | ❑属性 | ❑索引 |
| | ❑构造函数 | ❑事件 |
| | ❑析构函数 | |

**说明** 类是逻辑相关的数据和函数的封装，通常代表真实世界中的或概念上的事物。

## 4.2 程序和类：一个快速的示例

一个运行中的C#程序是一组相互作用的类型对象，它们中的大部分是类的实例。例如，假设有一个模拟扑克牌游戏的程序。当程序运行时，它有一个名为Dealer的类实例，它的工作就是运行游戏。还有几个名为Player的类实例，它们代表游戏的玩家。

Dealer对象保存纸牌的当前状态和玩家数目等信息。它的动作包括洗牌和发牌。

Player类有很大不同。它保存玩家名称等信息，并实现如分析玩家当前手上的牌和出牌这样的动作。运行中的程序如图4-1所示。

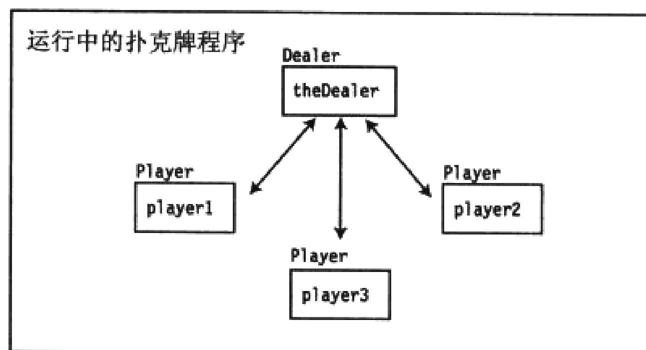



图4-1 一个正在运行的程序中的对象

一个真正的程序无疑会包含除Dealer和Player之外的许多其他的类，还会包括像Card和Deck这样的类。每一个类都模拟某种扑克牌游戏中的事物。

**说明** 运行中的程序是一组相互作用的对象。

## 4.3 声明类

或许你能猜到，虽然类型int、double和char由C#定义，但像Dealer和Player这样的类不是由语言定义的。如果想在程序中使用它们，你必须自己定义，通过编写类的声明定义类。

类的声明定义新类的特征和成员。它并不创建类的实例，但创建了用于创建实例的模板。类的声明提供下列内容：

- 类的名称
- 类的成员
- 类的特征

下面是一个最简单的类声明语法示例。大括号内包含了成员的声明，它们组成了类主体。类成员可以在类主体内部以任何顺序声明。这意味着成员的声明完全可以引用另一个在后面的类声明中才定义的成员。

```
关键字     类名
  ↓         ↓
class MyExcellentClass
{
   成员声明
}
```

例如，下面的代码给出了两个类声明的概貌：

```
class Dealer                              // 类声明
{
   ...
}

class Player                              // 类声明
{
   ...
}
```

**说明** 因为类声明"定义"了一个新类，所以经常会在文献和程序员的日常使用中看到类声明被称为"类定义"。

## 4.4 类成员

字段和方法是最重要的类成员类型。字段是数据成员，方法是函数成员。

### 4.4.1 字段

字段是隶属于类的变量。

- 它可以是任何类型，无论是预定义类型还是用户定义类型。
- 和所有变量一样，字段用来保存数据，并具有如下特征：
  - 它们可以被写入；
  - 它们可以被读取。

声明一个字段最简单的语句如下：

```
类型
↓
Type Identifier;
        ↑
     字段名称
```

例如，下面的类包含字段MyField的声明，它可以保存int值：

```
class MyClass
{  类型
    ↓
   int MyField;
          ↑
        字段名称
}
```

**说明** 与C和C++不同，C#在类型的外部不能声明全局变量（也就是变量或字段）。所有的字段都属于类型，而且必须在类型声明内部声明。

### 4.4.2 显式和隐式字段初始化

因为字段是一种变量，所以字段初始化语句在语法上和上一章所述的变量初始化语句相同。

- 字段初始化是字段声明的一部分，由一个等于号后面跟着一个求值表达式组成；
- 初始化值必须是编译期可决定的；

```
class MyClass
{
   int F1 = 17;
}         ↑
       字段初始值
```

- 如果没有初始化语句，字段的值会被编译器设为默认值。默认值由字段的类型决定。简单类型的默认值见表3-1（第3章）。可是总结起来，每种类型的默认值都是0，bool型是false，引用类型默认为null。

例如，下面的代码声明了4个字段，前两个字段被隐式初始化，另外两个字段被初始化语句显式初始化。

```
class MyClass
{
    int    F1;              // 初始化为0    - 值类型
    string F2;              // 初始化为null - 引用类型

    int    F3 = 25;         // 初始化为25
    string F4 = "abcd";     // 初始化为"abcd"
}
```

### 4.4.3　声明多个字段

可以通过用逗号分隔名称的方式，在同一条语句中声明多个相同类型的字段。但不能在一个声明中混合不同的类型。例如，可以把之前的4个字段声明结合成两条语句，并有相同的语义结果。

```
int    F1, F3 = 25;
string F2, F4 = "abcd";
```

### 4.4.4　方法

方法是具有名称的可执行代码块，可以从程序的很多不同地方执行，甚至从其他程序中执行。还有匿名方法，它们未被命名——它们将在第15章阐述。

当方法被调用（call 或 invoke）时，它执行自己所含的代码，然后返回到调用它的代码。有些方法返回一个值到它们被调用的位置。方法相当于C++中的成员函数。

声明方法的最简语法包括以下成分。

- 返回类型：它声明了方法返回值的类型。如果一个方法不返回值，那么返回类型被指定为void。
- 名称：这是方法的名称。
- 参数列表：它由至少一对空的圆括号组成。如果有参数（参数将在下一章阐述），它们被列在圆括号中间。
- 方法体：它由一对大括号组成，大括号内包含执行代码。

例如，下面的代码声明了一个类，带有一个名称为PrintNums的简单方法。从这个声明中可以看出下面几点关于PrintNums的情况：

- 它不返回值，因此返回类型指定为void；
- 它有空的参数列表；
- 它的方法体有两行代码。

```
class SimpleClass
{
 返回类型          参数列表
   ↓                 ↓
   void PrintNums (  )
   {
      Console.WriteLine("1");
      Console.WriteLine("2");
   }
}
```

**说明**　与C和C++不同，没有全局函数（也就是方法或函数）声明在类型声明的外部。

## 4.5 创建变量和类的实例

类的声明只是用于创建类的实例的蓝图。一旦类被声明，就可以创建类的实例。

- 类是引用类型，正如你从上一章学到的，这意味着它们要为数据引用和实际数据两者都申请内存。
- 数据的引用保存在一个类类型的变量中。所以，要创建类的实例，需要从声明一个类类型的变量开始。如果变量没有被初始化，它的值是未定义的。

图4-2阐明了如何定义保存引用的变量。左边顶端的代码是类Dealer的声明，下面是类Program的声明，它包含Main方法。Main声明了Dealer类型的变量theDealer，因为变量没有初始化，它的值是未定义的，如图4-2的右边所示。

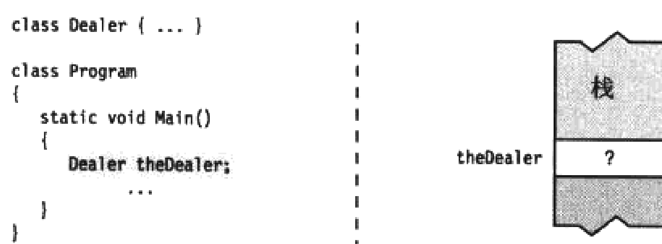
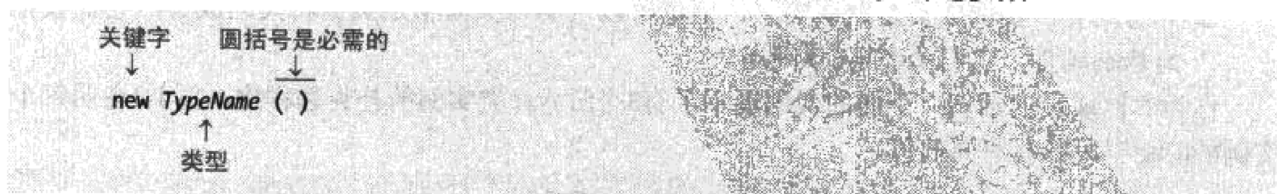



图4-2 为类变量的引用分配内存

## 4.6 为数据分配内存

声明类类型的变量所分配的内存是用来保存引用的，而不是用来保存类对象实际数据的。要为实际数据分配内存，需要使用new运算符。

- new运算符为任意指定类型的实例分配并初始化内存。它依据类型的不同从栈或堆里分配。
- 使用new运算符组成一个对象创建表达式，它的组成如下：
  - 关键字new。
  - 要分配内存的实例的类型名称。
  - 成对的圆括号，可能包括参数或没有参数，以后会进一步讨论参数。




- 如果内存分配给一个引用类型，则对象创建表达式返回一个引用，指向在堆中被分配并初始化的对象实例。

要分配和初始化用于保存类实例数据的内存，需要做的工作就是这些。使用new运算符创建对象创建表达式，并把它的返回值赋给类变量。这里有个示例：

```
Dealer theDealer;              // 声明引用变量
theDealer = new Dealer();      // 为类对象分配内存
            ↑
        对象创建表达式
```

图4-3左边的代码展示了用于分配内存并创建类Dealer实例的new运算符，随后实例被赋值给类变量。内存结构被阐明在图4-3中代码的右边。

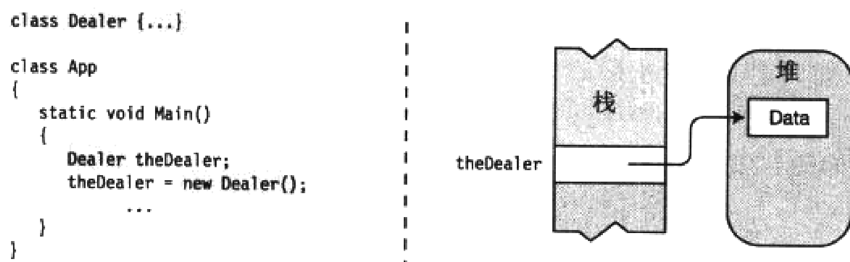



图4-3　为类变量的数据分配内存

### 结合步骤

两个步骤可以通过使用对象创建表达式初始化变量的方法结合到一起。

```
      声明变量
         ↓
Dealer theDealer = new Dealer();          // 声明并初始化
                          ↑
              使用对象创建表达式初始化变量
```

如果是本地变量而不是字段，可以让编译器推断左边声明部分的类型，以进一步简化语法，5.2节将会阐述这部分内容。

## 4.7　实例成员

类声明相当于蓝图，通过这个蓝图想创建多少个类的实例都可以。

- ❑ 实例成员：类的每个实例都是不同的实体，它们有自己的一组数据成员，不同于同一类的其他实例。因为这些数据成员都和类的实例相关，所以被称为实例成员。
- ❑ 静态成员：实例成员是默认类型，但也可以声明与类而不是实例相关的成员，它们被称为静态成员，它们会在第6章阐述。

作为实例成员的示例，下面的代码展示了有3个Player类实例的扑克牌程序。图4-4表明每个实例的Name字段都有不同的值。

```
class Dealer { ... }                     // 声明类
class Player {                           // 声明类
   string Name;                          // 字段
      ...
}
```

```
class Program {
   static void Main()
   {
      Dealer theDealer = new Dealer();
      Player player1 = new Player();
      Player player2 = new Player();
      Player player3 = new Player();
      ...
   }
}
```

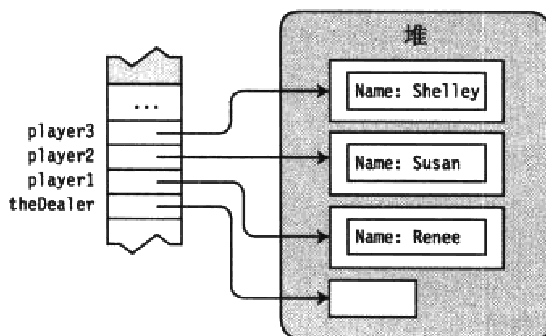



图4-4 实例成员在类对象之间是不同的

## 4.8 访问修饰符

从类的内部，任何函数成员都可以使用成员的名称访问类中任意的其他成员。

访问修饰符是成员声明的可选部分，指明程序的其他部分如何访问成员。访问修饰符放在简单声明形式之前。下面是字段和方法声明的语法：

```
字段
  访问修饰符 类型 标识符;

方法
  访问修饰符 返回类型 方法名()
  {
     ...
  }
```

5种成员访问控制如下。本章将阐述前两种，其他的在第7章阐述。

- ❑ 私有的（private）
- ❑ 公有的（public）
- ❑ 受保护的（protected）
- ❑ 内部的（internal）
- ❑ 受保护内部的（protected internal）

## 私有访问和公有访问

私有成员只能从声明它的类的内部访问，其他的类不能看见或访问它们。

- 私有访问是默认的访问级别，所以，如果一个成员在声明时不带访问修饰符，那它就是私有成员。
- 还可以使用private访问修饰符显式地声明一个成员为私有。
- 隐式地声明私有成员和显式地声明没有语义上的不同，两种形式是等价的。

例如，下面的两个声明都指定了private int成员：

```
        int MyInt1;                    // 隐式声明为私有
private int MyInt2;                    // 显式声明为私有
   ↑
访问修饰符
```

公有成员可以被程序中的所有其他对象访问。必须使用public访问修饰符指定公有访问。

```
访问修饰符
   ↓
public int MyInt;
```

### 1. 公有访问和私有访问图示

本文中的插图把类表示为标签框，如图4-5所示。

- 类成员被表示为类框中的小标签框。
- 私有成员被表示为完全封闭在它们的类框中。
- 公有成员被表示为部分伸出到它们的类框之外。

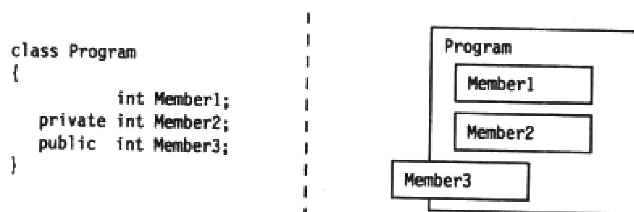



图4-5　表示类和成员

### 2. 成员访问示例

类C1声明了公有和私有的字段和方法，图4-6阐明了类C1的成员的可见性。

```
class C1
{
   int         F1;                    // 隐式私有字段
   private int F2;                    // 显式私有字段
   public  int F3;                    // 公有字段

   void DoCalc()                      // 隐式私有字段
   {
      ...
```

```
    }

    public int GetVal()                    // 公有方法
    {
       ...
    }
}
```

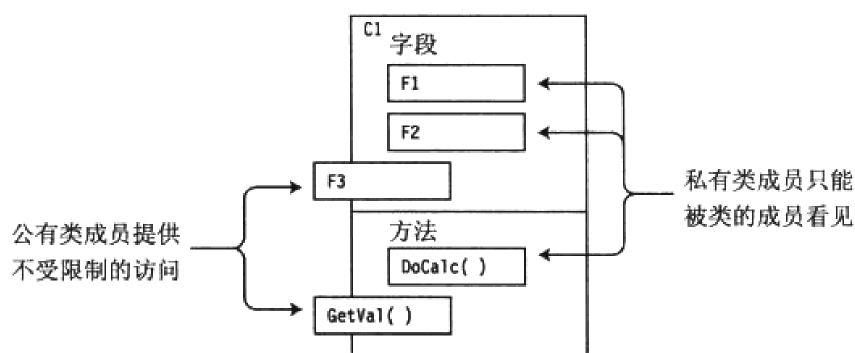



图4-6 类的私有成员和公有成员

## 4.9 从类的内部访问成员

如前所述，类的成员仅用其他类成员的名称就可以访问它们。

例如，下面的类声明展示了类的方法对字段和其他方法的访问。即使字段和两个方法被声明为private，类的所有成员还是都可以被类的任何方法（或任何函数成员）访问。图4-7阐明了这段代码。

```
class DaysTemp
{
   // 字段
   private int High = 75;
   private int Low  = 45;

   // 方法
   private int GetHigh()
   {
      return High;                        // 访问私有字段
   }

   private int GetLow()
   {
      return Low;                         // 访问私有字段
   }

   public float Average ()
   {
      return (GetHigh() + GetLow()) / 2;     // 访问私有方法
   }           ↑           ↑
             访问私有方法
}
```

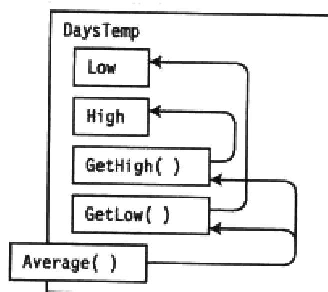



图4-7 类内部的成员可以自由地互相访问

## 4.10 从类的外部访问成员

要从类的外部访问实例成员，必须包括变量名称和成员名称，中间用句点（.）分隔。这称为点运算符，将会在以后更详细地讨论它。

例如，下面代码的第二行展示了一个从类的外部访问方法的示例：

```
DaysTemp myDt = new DaysTemp();        // 创建类的对象
float fValue  = myDt.Average();        // 从外部访问
                  ↑      ↑
              变量名称  成员名称
```

举个例子，下面的代码声明了两个类：DaysTemp和Program。

- DaysTemp内的两个字段被声明为public，所以它们可以被从类的外部访问。
- 方法Main是类Program的成员。它创建了一个变量和类DaysTemp的对象，并赋值给对象的字段。然后它读取字段的值并打印出来。

```
class DaysTemp                                          // 声明类DaysTemp
{
   public int High = 75;
   public int Low = 45;
}

class Program                                           // 声明类Program
{
   static void Main()
   {           变量名称
                  ↓
      DaysTemp temp = new DaysTemp();                   // 创建对象
     变量名称和字段
          ↓
      temp.High = 85;                                   // 字段赋值
      temp.Low  = 60;               变量名称和字段
                                          ↓
      Console.WriteLine("High:  {0}", temp.High );      // 读取字段值
      Console.WriteLine("Low:   {0}", temp.Low  );
   }
}
```

这段代码产生如下输出：

```
High:  85
Low:   60
```

## 4.11 综合应用

下面的代码创建两个实例并把它们的引用保存在名称为t1和t2的变量中。图4-8阐明了内存中的t1和t2。这段代码示范了使用类的3种行为：

- 声明一个类；
- 创建类的实例；
- 访问类的成员（也就是写入字段和读出字段）。

```
class DaysTemp                              // 声明类
{
   public int High, Low;                    // 声明实例字段
   public int Average()                     // 声明实例方法
   {
      return (High + Low) / 2;
   }
}

   class Program
   {
      static void Main()
      {
         // 创建两个DaysTemp实例
         DaysTemp t1 = new DaysTemp();
         DaysTemp t2 = new DaysTemp();

         // 给字段赋值
         t1.High = 76; t1.Low = 57;
         t2.High = 75; t2.Low = 53;

         // 读取字段值
         // 调用实例的方法
         Console.WriteLine("t1: {0}, {1}, {2}",
                                    t1.High, t1.Low, t1.Average() );
         Console.WriteLine("t2: {0}, {1}, {2}",
                                    t2.High, t2.Low, t2.Average() );
                                       ↑        ↑         ↑
      }                               字段      字段       方法
   }
```

这段代码的输出如下：

```
t1: 76, 57, 66
t2: 75, 53, 64
```

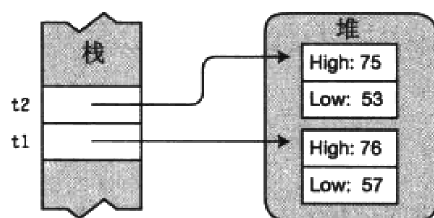



图4-8 实例t1和t2的内存布局

第5章

# 方　　法

本章内容

- 方法的结构
- 本地变量
- 本地常量
- 方法调用
- 返回值
- 参数
- 值参数
- 引用参数
- 输出参数
- 参数数组
- 参数类型总结
- 栈帧
- 递归
- 方法重载

## 5.1　方法的结构

本质上，方法是一块具有名称的代码。可以使用方法的名称执行代码，也可以把数据传入方法并接收数据输出。

如前面各章所述，方法是类的函数成员。方法有两个主要部分，如图5-1所示：方法头和方法体。

- 方法头指定方法的特征，包括：
  - 方法是否返回数据，如果返回，返回什么类型；
  - 方法的名称；
  - 什么类型的输入可以传入方法。
- 方法体包含可执行代码的语句序列。执行从方法体的第一条语句开始，一直到整个方法结束。

```
void DoTask ( )      ← 方法头
{
      语句1
      语句2
        ⋮           方法体
      语句x
}
```

图5-1　方法的结构

下面的示例展示了方法头的形式。接下来阐述其中的每一部分。

```
int MyMethod ( int intpar1, string strpar1 )
 ↑      ↑                 ↑
返回   方法              参数
类型   名称              列表
```

方法还可以是另一种称为结构（struct）的用户自定义类型的函数成员。结构将在第12章阐述。本章涉及的大部分关于类方法的内容也适用于结构方法。

例如，下面的代码展示了一个名称为MyMethod的简单方法，它多次轮流调用WriteLine方法。

```
void MyMethod()
{
   Console.WriteLine("First");
   Console.WriteLine("Last");
}
```

## 方法体内部的代码执行

方法体是一个块，是大括号括起的语句序列（可以参照第2章）。块可以包含以下项目：

- 本地变量
- 控制流结构
- 方法调用
- 内嵌的块

图5-2展示了一个方法体及其组成的示例。

```
static void Main()
{
   int myInt = 3;          ← 本地变量

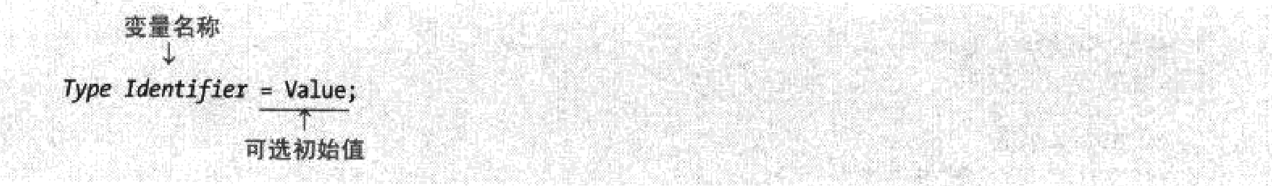
   while (myInt > 0)       ┐
   {                       │
      --myInt;             │ 控制流结构
      PrintMyMessage();    │
   }                       ┘
}          ↖ 方法调用
```

图5-2　方法体示例

## 5.2 本地变量

和字段一样，本地变量也保存数据。字段通常保存和对象状态有关的数据，而本地变量用于保存本地的或临时的计算数据。表5-1对比了本地变量和实例字段的差别。

下面这行代码展示了本地变量声明的语法。可选的初始化语句由等号后面跟着用于初始化变量的值组成。




- ❑ 本地变量的存在性仅限于创建它的块及其内嵌的块。
  - 它从声明它的那一点开始存在。
  - 它在块完成执行时结束存在。
- ❑ 可以在方法体内任意位置声明本地变量。

下面的示例展示了两个本地变量的声明和使用。第一个是int类型变量，第二个是SomeClass类型变量。

```
static void Main( )
{
   int myInt     = 15;
   SomeClass sc = new SomeClass();
   ...
}
```

**表5-1 对比实例字段和本地变量**

| | 实例字段 | 本地变量 |
|---|---|---|
| **生存期** | 从实例被创建时开始直到实例不再被访问时结束 | 从它在块中被声明的那点开始在块完成执行时结束 |
| **隐式初始化** | 初始化成该类型的默认值 | 没有隐式初始化。如果变量在使用之前没有被赋值，编译器就会产生一条错误信息 |
| **存储区域** | 类的所有字段都存储在堆里，无论它们是值类型的还是引用类型的 | 值类型：存储在栈里<br>引用类型：引用存储在栈里，数据存储在堆里 |

### 5.2.1 类型推断和 var 关键字

如果观察下面的代码，你会发现当你在声明的开始部分提供类型名时，你提供的是编译器能从初始化语句的右边推断出来的信息。

- ❑ 在第一个变量声明中，编译器能推断出15是int型。
- ❑ 在第二个声明中，右手边的对象创建表达式返回了一个MyExcellentClass类型的对象。

所以在两种情况中，在声明的开始部分包括显式的类型名是多余的。

```
static void Main( )
{
    int total = 15;
    MyExcellentClass mec = new MyExcellentClass();
    ...
}
```

从C# 3.0开始，可以在变量声明的开始部分的显式类型名的位置使用新的关键字var，如：

```
static void Main( )
{  关键字
    ↓
    var total = 15;
    var mec   = new MyExcellentClass();
    ...
}
```

var关键字并不是某种特别类型的符号。它只是句法上的速记，表示任何可以从初始化的右边推断出的类型。在第一个声明中，它是int的速记；在第二个声明中，它是MyExcellentClass的速记。前文中使用显式类型名的代码片段和使用var关键字的代码片段是语义等价的。

使用var关键字有一些重要的条件：

- 只能用于本地变量，不能用于字段。
- 只能在变量声明中包含初始化时使用。
- 一旦编译器推断出类型，它就是固定且不能更改的。

**说明** var关键字不像JavaScript的var那样可以引用不同的类型。它是从等号右边推断出的实际类型的速记。var关键字并不改变C#的强类型性质。

### 5.2.2 嵌套块中的本地变量

方法体内部可以嵌套其他的块。

- 可以有任意数量的块，并且它们既可以是顺序的也可以更深层嵌套的。块可以嵌套到任何级别。
- 本地变量可以在嵌套块的内部声明，并且和所有的本地变量一样，它们的生存期仅限于声明它们的块及其内嵌块。

图5-3阐明了两个本地变量的生存期，展示了代码和栈的状态。箭头标出了刚执行过的行。

- 变量var1声明在方法体中，在嵌套块之前。
- 变量var2声明在嵌套块内部。它从被声明那一刻开始存在，直到声明它的那个块的尾部结束。
- 当控制传出嵌套块时，它的本地变量被从栈中弹出。

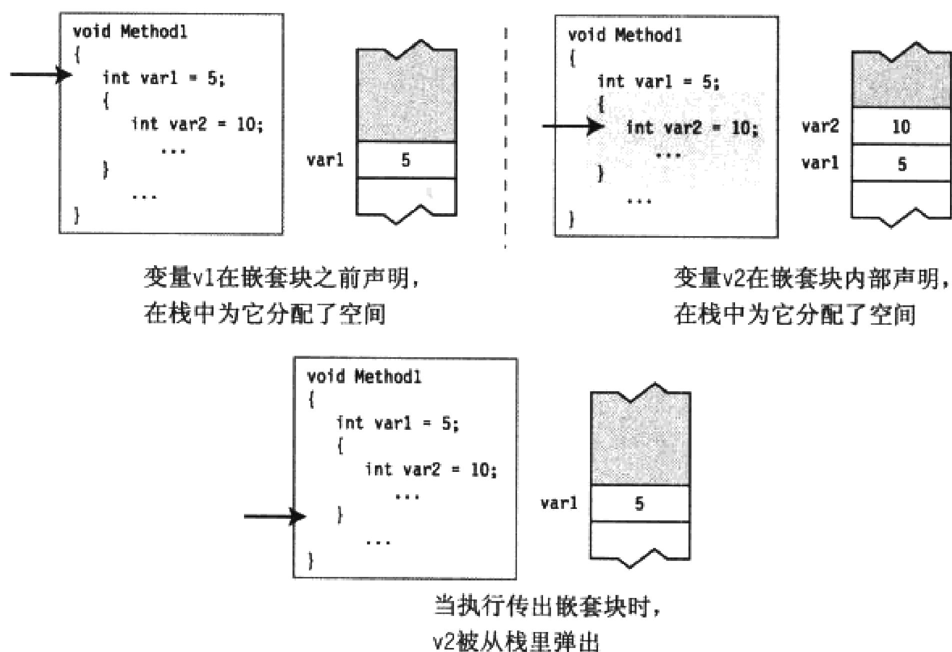



图5-3 本地变量的生存期

**说明** 在C和C++中，可以先定义一个本地变量，然后在嵌套块中定义另一个相同名称的本地变量。在内部范围，内部名称掩盖了外部名称。然而，在C#中不管嵌套级别如何，都不能在第一个名称的有效范围内声明另一个同名的本地变量。

## 5.3 本地常量

本地常量很像本地变量，只是一旦它被初始化，它的值就不能被改变。如同本地变量，本地常量必须声明在块的内部。

常量的两个最重要的特征如下：

- 常量在声明中必须初始化。
- 常量在声明后不能改变。

常量的核心声明如下所示。语法与字段或变量的声明相同，除了下面内容：

- 在类型之前增加关键字const。
- 必须有初始化语句。初始化值必须在编译期决定，通常是一个预定义简单类型或由其组成的表达式。它还可以是null引用，但它不能是某对象的引用，因为对象的引用是在运行时决定的。

**说明** 关键字const不是一个修饰符，而是核心声明的一部分。它必须直接放在类型的前面。

```
关键字
  ↓
const Type Identifier = Value;
                        ↑
                  初始化值是必需的
```

就像本地变量，本地常量声明在方法体或代码块里，并在声明它的块结束的地方失效。例如，在下面的代码中，本地常量PI在方法DisplayRadii结束后失效。

```
void DisplayRadii()
{
   const double PI = 3.1416;                    // 声明本地常量

   for (int radius = 1; radius <= 5; radius++)
   {
      double area = radius * radius * PI;       // 读取本地常量
      Console.WriteLine
         ("Radius: {0}, Area: {1}", radius, area);
   }
}
```

## 控制流

方法包含了大部分组成程序行为的代码。剩余部分在其他的函数成员中，如属性和运算符，但大多数在方法中。

术语控制流指的是程序从头到尾的执行流程。默认情况下，程序执行持续地从一条语句到下一条语句，控制语句允许你改变执行的顺序。

在这一节，只会提及一些能用于代码的有用的控制语句，第9章会详细阐述它们。

- 选择语句：这些语句可以选择哪条语句或语句块来执行。
  - if：有条件地执行一条语句；
  - if...else：有条件地执行一条或另一条语句；
  - switch：有条件地执行一组语句中的一条。
- 迭代语句：这些语句可以在一个语句块上循环或迭代。
  - for：循环——在顶部测试；
  - while：循环——在顶部测试；
  - do：循环——在底部测试；
  - foreach：为一组中每个成员执行一次。
- 跳转语句：这些语句可以让你从代码块或方法体内部的一个地方跳到另一个地方。
  - break：跳出当前循环；
  - continue：到当前循环的底部；
  - goto：到一个命名的语句；
  - return：返回到发起调用的方法。

例如，下面的代码展示了两个控制流语句，先别管那些细节。

```
void SomeMethod()
{
   int intVal = 3;
          相符比较操作符
               ↓
   if( intVal == 3 )                            // if语句
      Console.WriteLine("Value is 3.");

   for( int i=0; i<5; i++ )                     // for语句
      Console.WriteLine("Value of i: {0}", i);
}
```

## 5.4 方法调用

可以从方法体的内部调用其他方法。

- 英文中call（调用）方法和invoke方法是同义的。
- 调用方法时要使用方法名并带上参数列表。参数列表将稍后讨论。

例如，下面的类声明了一个名称为PrintDateAndTime的方法，它被从方法Main的内部调用。

```
class MyClass
{
   void PrintDateAndTime( )                  // 声明方法
   {
      DateTime dt = DateTime.Now;            // 获取当前日期和时间
      Console.WriteLine("{0}", dt);          // 输出
   }

   static void Main()                        // 声明方法
   {
      MyClass mc = new MyClass();
      mc.PrintDateAndTime( );                // 调用方法
   }              ↑          ↑
}             方法名称    空参数列表
```

图5-4阐明了调用方法时的动作顺序：

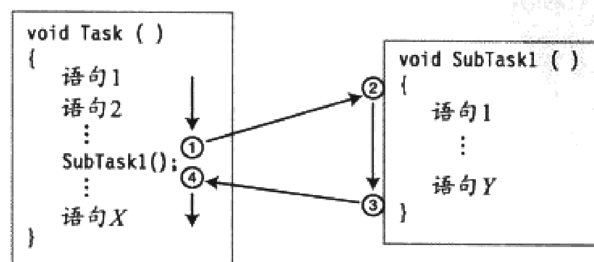



图5-4 调用方法时的控制流

(1) 当前方法的执行在调用点被挂起。
(2) 控制转移到被调用方法的开始。
(3) 被调用方法执行直到完成。
(4) 控制回到发起调用的方法。

## 5.5 返回值

方法可以向调用代码返回一个值。返回的值被插入到调用代码中发起调用的表达式所在的位置。

- 要返回值，方法必须在方法名前面声明一个**返回类型**。
- 如果方法不返回值，它必须声明void返回类型。

下面的代码展示了两个方法声明。第一个返回int型值，第二个不返回值。

```
返回类型
 ↓
int GetHour()      { ... }
void DisplayHour() { ... }
 ↑
不返回值
```

声明了返回类型的方法必须使用下面形式的返回语句从方法中返回一个值。返回语句包括关键字return及其后面的表达式。每一条穿过方法的路径都必须以一条这种形式的return语句结束。

```
return Expression;                    // 返回一个值
            ↑
      计算返回类型的值
```

例如，下面的代码展示了一个名称为GetHour的方法，它返回int型值。

```
返回类型
 ↓
int GetHour( )
{
   DateTime dt = DateTime.Now;           // 获取当前日期和时间
   int hour = dt.Hour;                   // 获取小时数

   return hour;                          // 返回一个int
}        ↑
     返回语句
```

也可以返回用户定义类型的对象。例如，下面的代码返回一个类型MyClass的对象。

```
返回类型——MyClass
   ↓
MyClass method3( )
{
```

```
    MyClass mc = new MyClass();
       ...
    return mc;                                 // 返回一个Myclass对象
}
```

另一个示例。在下面的代码中，方法GetHour在Main的WriteLine语句中被调用，并在该位置返回一个int值到WriteLine语句中。

```
class MyClass
{      ↓ 返回类型
   public int GetHour()
   {
      DateTime dt = DateTime.Now;              // 获取当前日期和时间
      int hour = dt.Hour;                      // 获取小时数

      return hour;                             // 返回一个int
   }          ↑
}           返回值


class Program
{
   static void Main()
   {                                  方法调用
      MyClass mc = new MyClass();    ____↓____
      Console.WriteLine("Hour: {0}", mc.GetHour());
   }                                  ↑    ↑
}                              实例名称  方法名称
```

## 返回语句和 void 方法

在上一节，我们看到有返回值的方法必须包含返回语句。void方法不需要返回语句。当控制流到达方法体的关闭大括号时，控制返回到调用代码，并且没有值被插入到调用代码中。

不过，当特定条件符合的时候，我们常常会提前退出方法以简化程序逻辑。

- 可以在任何时候使用下面形式的返回语句退出方法，不带参数：

```
return;
```

- 这种形式的返回语句只能用于声明为void类型的方法。

例如，下面的代码展示了一个名称为SomeMethod的void方法的声明。它可以在三个可能的地方返回到调用代码。前两个地方在if语句的分支内。if语句将在第9章阐述。最后一个方法在方法体的结尾处。

```
void返回类型
   ↓
  void SomeMethod()
  {
```

```
    ...
    If ( SomeCondition )
       return;                                 // 返回到调用代码
    ...

    If ( OtherCondition )
       return;                                 // 返回到调用代码

    ...
}                                              // 返回到调用代码
```

下面的代码展示了一个带有一条返回语句的void方法示例。该方法只有当时间是下午的时候才写出一条消息，如图5-5所示，其过程如下：

- 首先，方法获取当前日期和时间（现在不用理解这些细节）。
- 如果小时数小于12（也就是上午），返回语句被执行，并且控制立刻返回到调用方法。
- 如果小时数是12或更大，返回语句被跳过，WriteLine语句被执行。

```
class MyClass
{    ↓ void返回类型
   void TimeUpdate() {
   DateTime dt = DateTime.Now;                 // 获取当前日期和时间
      if (dt.Hour < 12)                        // 若小时数小于12
         return;                               // 则返回
           ↑
      返回到调用方法
      Console.WriteLine("It's afternoon!");    // 否则，输出消息
   }

   static void Main() {
      MyClass mc = new MyClass();              // 创建一个类实例
      mc.TimeUpdate();                         // 调用方法
   }
}
```

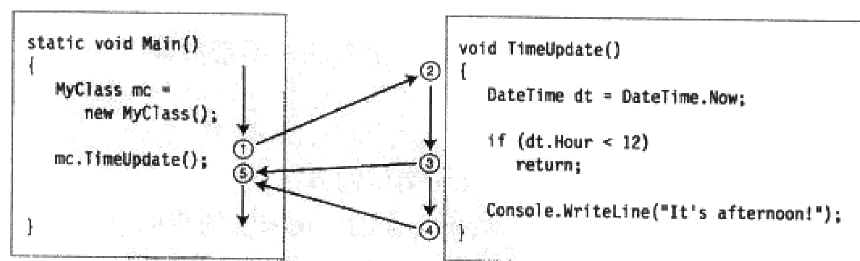



图5-5　使用void返回类型的返回语句

## 5.6　参数

迄今为止，你已经看到方法是可以被程序中很多地方调用的命名代码单元，它能把一个值返回给调用代码。返回一个值的确有用，但如果需要返回多个值呢？还有，能在方法开始执行的时

候把数据传入方法也会有用。参数就是允许你做这两件事的特殊变量。

## 5.6.1 形参

形参是声明在方法的参数列表中而不是方法体中的本地变量。

下面的方法头展示了参数声明的语法。它声明了两个形参：一个是int型，另一个是float型。

```
public void PrintSum( int x, float y )
{ ... }                    ↑
                       形参声明
```

- 因为形参是变量，所以它们有类型和名称，并能被写入和读出。
- 和方法中的其他变量不同，参数在方法体的外面定义并在方法开始之前初始化。但有一种类型例外，我们将很快谈到它。
- 参数列表中可以有任意数目的形参声明，而且声明必须用逗号隔开。

形参在整个方法体内使用，在大部分地方就像其他本地变量一样。例如，下面方法PrintSum的声明使用两个形参x和y，以及一个本地变量Sum，它们都是int型。

```
public void PrintSum( int x, int y )
{
   int Sum = x + y;
   Console.WriteLine("Newsflash:  {0} + {1} is {2}", x, y, Sum);
}
```

## 5.6.2 实参

当代码调用一个方法时，形参的值必须在方法的代码开始执行之前被初始化。

- 用于初始化形参的表达式或变量称为实参。
- 实参放在方法调用的参数列表中。

例如，下面的代码展示了方法PrintSum的调用，它有两个int型的实参。

```
PrintSum( 5, SomeInt );
          ↑     ↑
      表达式  int类型变量
```

当方法被调用的时候，每个实参的值都被用于初始化相应的形参，方法体随后被执行。图5-6阐明了实参和形参的关系。

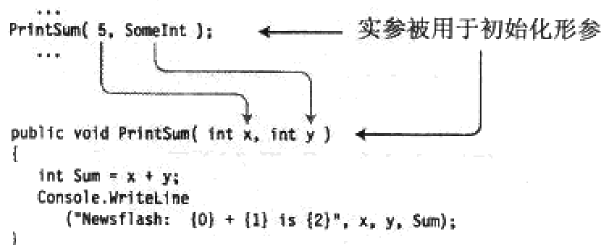



图5-6 实参初始化对应的形参

调用方法时，必须满足下列要求：

- 实参的数目必须和形参的数目相同（有一种情况例外，不久将会讨论到）。
- 每个实参必须和相应形参的类型匹配。

### 5.6.3　带输入参数的方法示例

在下面的代码中，类MyClass声明了两个方法：一个方法接受两个整数并返回它们的和，另一个方法接受两个float并返回它们的平均值。

```
class MyClass                 形参
{                              ↓
   public int Sum(int x, int y)                            // 声明方法
   {
      return x + y;                                        // 返回和
   }                                 形参
                                      ↓
   public float Avg(float Input1, float Input2)            // 声明方法
   {
      return (Input1 + Input2) / 2.0F;                     // 返回平均值
   }
}


class Class1
{
   static void Main()
   {
      MyClass MyT = new MyClass();
      int SomeInt = 6;

      Console.WriteLine
         ("Newsflash:  Sum: {0} and {1} is {2}",
             5, SomeInt, MyT.Sum( 5, SomeInt ));             // 调用方法
                                      ↑
      Console.WriteLine              实参
         ("Newsflash:  Avg: {0} and {1} is {2}",
             5, SomeInt, MyT.Avg( 5, SomeInt ));             // 调用方法
   }                                  ↑
}                                    实参
```

这段代码产生如下输出：

```
Newsflash:  Sum: 5 and 6 is 11
Newsflash:  Avg: 5 and 6 is 5.5
```

## 5.7　值参数

有几种参数，它们使用稍微不同的方法从方法传入或传出数据。你到现在一直看到的这种类

型是默认的类型，称为值参数。

使用值参数，数据通过复制实参的值到形参的方式传递到方法。方法被调用时，系统做如下操作：

- 在栈中为形参分配空间。
- 复制实参到形参。

一个值参数的实参不一定是变量。它可以是任何能计算成相应数据类型的表达式。例如，下面的代码展示了两个方法调用。在第一个方法调用中，实参是float类型的变量；在第二个方法调用中，它是计算成float的表达式。

```
float func1( float Val )                              // 声明方法
{ ... }        float数据类型

{
   float j = 2.6F;
   float k = 5.1F;  float类型变量

   float fValue1 = func1( k );                        // 方法调用
   float fValue2 = func1( (k + j) / 3 );              // 方法调用
   ...                 计算成float的表达式
```



变量在用作实参前必须被赋值（在作为输出参数时除外，不久将会讨论到输出参数）。对于引用类型，变量可以被赋值为一个引用或null。

第3章阐述了值类型，还记得吗，它是含有它们自己的数据的类型。不要弄混了，我现在正在谈论值参数，它们完全不同。请记住，值参数是把实参的值复制到形参的参数。

例如，下面的代码展示了一个名称为MyMethod的方法，它有两个参数：一个MyClass型变量和一个int。

- 方法给类的字段和那个int都加上5。
- 你可能还注意到MyMethod使用了修饰符static，至今还没有解释它，现在你可以忽略它，我会在第6章谈论static方法。

```
class MyClass
{ public int Val = 20; }                        // 初始化字段为20

class Program                   形参
{
   static void MyMethod( MyClass f1, int f2 )
   {
      f1.Val = f1.Val + 5;                      // 参数的字段加5
      f2     = f2 + 5;                          // 另一参数加5
   }

   static void Main( )
   {
```

```
        MyClass A1 = new MyClass();
        int     A2 = 10;

        MyMethod( A1, A2 );                    // 调用方法
    }                ↑
}                   实参
```

图5-7阐明了实参和形参在方法执行的不同阶段时的值，它表明：

- 在方法被调用前，用作实参的变量A2和A2已经在栈里了。
- 随着方法的开始，系统在栈中为形参分配空间，并从实参复制值。
  - 因为A1是引用类型的，所以引用被复制，结果实参和形参都引用堆中的同一个对象。
  - 因为A2是值类型的，所以值被复制，产生了一个独立的数据项。
- 在方法的结尾，f2和对象f1的字段都被加上了5。
- 方法执行后，形参被从栈中弹出。
  - A2，值类型，它的值不受方法行为的影响。
  - A1，引用类型，但它的值被方法的行为改变了。

1. 调用前　　2. 方法开始　　3. 方法结束　　4. 方法之后

图5-7　值参数

## 5.8　引用参数

第二种参数类型称为引用参数。

- 使用引用参数时，必须在方法的声明和调用中都使用ref修饰符。
- 实参必须是变量，在用作实参前必须被赋值。如果是引用类型变量，可以赋值为一个引

用或null值。

例如，下面的代码阐明了引用参数的声明和调用的语法：

```
              包含ref修饰符
                   ↓
void MyMethod( ref int val )          // 方法声明
{ ... }

int y = 1;                            // 实参变量
MyMethod ( ref y );                   // 方法调用
              ↑
          包含ref修饰符

MyMethod ( ref 3+5 );                 // 出错了！
                 ↑
             必须使用变量
```

请记住，对于值参数，系统在栈里为形参分配内存。相应地，引用参数有以下特征：

- 不在栈中为形参分配新的内存。
- 形参的名称相当于实参变量的别名，引用与实参相同的内存位置。

既然形参名和实参名引用相同的内存位置，显然，在方法执行期间对形参所做的任何改变在方法完成之后通过实参变量都是可见的。

例如，下面的代码再次展示了方法MyMethod，但这一次参数是引用参数而不是值参数。

```
class MyClass
{ public int Val = 20; }                          // 初始化字段为20

class Program         ref修饰符     ref修饰符
{                         ↓             ↓
   static void MyMethod(ref MyClass f1, ref int f2)
   {
      f1.Val = f1.Val + 5;                        // 参数的字段加5
      f2 = f2 + 5;                                // 另一参数加5
   }

   static void Main()
   {
      MyClass A1 = new MyClass();
      int A2 = 10;

      MyMethod(ref A1, ref A2);                   // 调用方法
   }                ↑       ↑
}                   ref修饰符
```

图5-8阐明了在方法执行的不同阶段中实参和形参的值：

- 在方法调用之前，将要被用作实参的变量A1和A2已经在栈里了。
- 在方法的开始，形参名被设置为实参的别名。变量A1和f1引用相同的内存位置，A2和f2

引用相同的内存位置。

- 在方法的结束位置，f2和f1的对象的字段都被加上了5。
- 方法执行之后，形参的名称已经失效，但是A2的值和A1指向的对象的值都被方法内的行为改变了。A2是值类型变量，A1是引用类型。

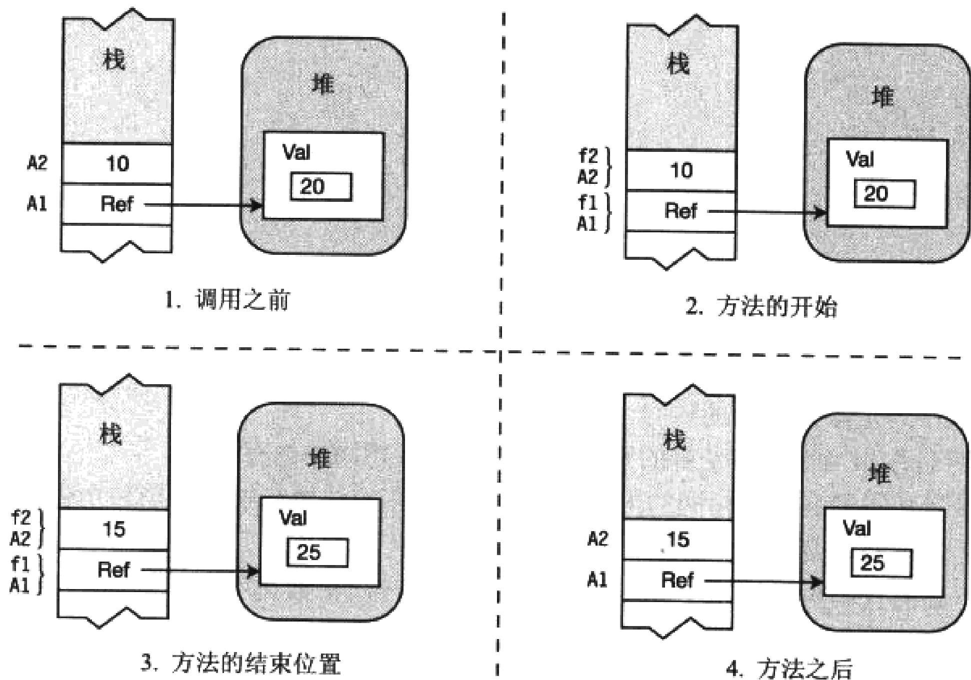



图5-8 引用参数

## 5.9 输出参数

输出参数用于从方法体内把数据传出到调用代码，它们非常类似引用参数。如同引用参数，输出参数有以下要求：

- 必须在声明的调用中都使用修饰符。输出参数的修饰符是out而不是ref。
- 实参必须是变量，不能是其他表达式类型。

例如，下面的代码声明了名称为MyMethod的方法，它带有单个输出参数。

```
               out修饰符
                  ↓
void MyMethod( out int val )          // 方法声明
{ ... }

...
int y = 1;                            // 实参变量
MyMethod ( out y );                   // 方法调用
             ↑
          out修饰符
```

就像引用参数，输出参数的形参担当实参的别名，形参和实参都是同一内存位置的名称。显然，方法内对形参的任何改变在方法之后通过实参变量都是可见的。

与引用参数不同，输出参数有以下要求：

- 在方法内部，输出参数在被读取之前必须被赋值。这意味着参数的初始值是无关的，而且没有必要在方法调用之前为实参赋值。
- 每个输出参数在方法返回之前必须被赋值。

因为方法内的代码在读出输出变量之前必须对其写入，所以不可能使用输出参数把数据传入方法。事实上，如果方法中有任何执行路径试图在输出参数被赋值之前读取它，编译器就会产生一条错误信息。

```
public void Add2( out int outValue )
{
   int var1 = outValue + 2;  // 出错了！在方法赋值之前
}                            // 无法读取输出变量
```

例如，下面的代码再次展示了方法MyMethod，但这次使用输出参数。

```
class MyClass
{ public int Val = 20; }                    // 字段初始化为20

class Program          out修饰符       out修饰符
{                          ↓               ↓
   static void MyMethod(out MyClass f1, out int f2)
   {
      f1 = new MyClass();                   // 创建一个类变量
      f1.Val = 25;                          // 赋值类字段
      f2     = 15;                          // 赋值int参数
   }

   static void Main()
   {
      MyClass A1 = null;
      int A2;

      MyMethod(out A1, out A2);             // 调用方法
   }             ↑       ↑
}              out修饰符
```

图5-9阐述了在方法执行的不同阶段中实参和形参的值：

- 在方法调用之前，将要被用作实参的变量A1和A2已经在栈里了。
- 在方法的开始，形参名被设置为实参的别名。变量A1和f1引用相同的内存位置，A2和f2引用相同的内存位置。名称A1和A2已经失效并且不能从MyMethod内部访问。
- 在方法内部，代码创建了一个MyClass类型的对象并把它赋值给f1。然后赋一个值给f1的字段，也赋一个值给f2。对f1和f2的赋值都是必需的，因为它们是输出参数。
- 方法执行之后，形参的名称已经失效，但是引用类型的A1和值类型的A2的值都被方法内的

行为改变了。

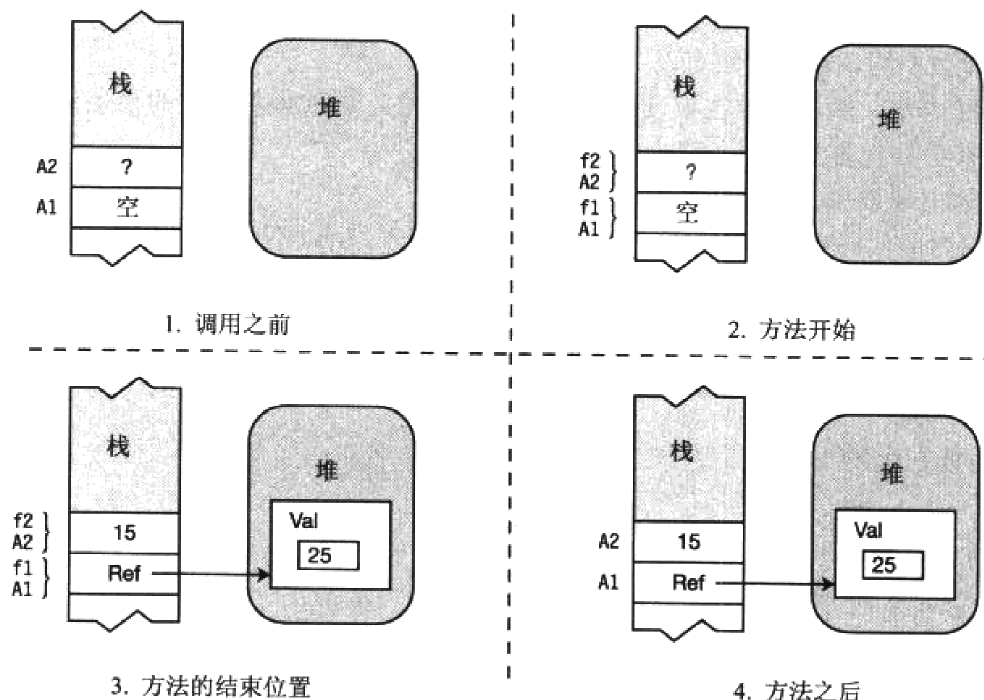



图5-9 输出参数

## 5.10 参数数组

至此，本书所述的参数类型都必须严格地一个实参对应一个形参。参数数组不一样，它允许零个或多个实参对应一个特殊的形参。参数数组的重点如下：

- 在一个参数列表中只能有一个参数数组。
- 如果有，它必须是列表中的最后一个。

声明一个参数数组必须做的事：

- 在数据类型前使用params修饰符。
- 在数据类型后放置一组空的方括号。

下面的方法头展示了int型参数数组的声明语法。在这个示例中，形参inVals可以代表零个或多个int实参。

```
                     int型参数数组
                          ↓
void ListInts( params int[] inVals )
{ ...             ↑            ↑
                修饰符        参数名称
```

类型名后面的空方括号指明了参数是一个整数数组。在这里不必在意数组的细节，它们将在第14章详细阐述。为了我们此处的目的，所有你需要了解的内容如下：

- 数组是一组整齐的相同类型的数据项。

❑ 数组使用一个数字索引进行访问。
❑ 数组是一个引用类型，因此它的所有数据项都保存在堆中。

## 5.10.1 方法调用

有两个方法提供实参，可以使用如下的形式：
❑ 一个逗号分隔的该数据类型元素的列表。所有元素必须是方法声明中指定的类型。

```
ListInts( 10, 20, 30 );              // 3个int
```

❑ 一个该数据类型元素的一维数组。

```
int[] intArray = {1, 2, 3};
ListInts( intArray );                    // 一个数组变量
```

请注意，在这些示例中，并不在调用时使用params修饰符。参数数组中修饰符的使用并不适用其他参数类型的模式。
❑ 其他参数类型要么使用修饰符，要么不使用修饰符。
- 值参数的声明和调用都不带修饰符。
- 引用参数和输出参数在两个地方都需要修饰符。

❑ 然而参数数组：
- 在声明中需要修饰符。
- 在调用中不允许有修饰符。

**延伸式**

方法调用的第一种形式有时被称为延伸式，这种形式在调用中使用分离的实参。

例如，下面代码中的方法ListInts的声明可以匹配它下面所有的方法调用，虽然它们有不同数目的实参。

```
void ListInts( params int[] inVals ) { ... }        // 方法声明

...
ListInts( );                                        // 0 实参
ListInts( 1, 2, 3 );                                // 3 实参
ListInts( 4, 5, 6,  7 );                            // 4 实参
ListInts( 8, 9, 10, 11, 12 );                       // 5 实参
```

在使用一个为参数数组分离实参的调用时，编译器做下面的事：
❑ 接受实参列表，用它们在堆中创建并初始化一个数组。
❑ 把数组的引用保存到栈中的形参里。
❑ 如果在对应的形参数组的位置没有实参，编译器会创建一个有零个元素的数组来使用。

例如，下面的代码声明了一个名称为ListInts的方法，它带有一个参数数组。Main声明了三

个整数并把它们传给了数组。

```
class MyClass                          参数数组
{                                         ↓
   public void ListInts( params int[] inVals )
   {
      if ( (inVals != null) && (inVals.Length != 0))
         for (int i = 0; i < inVals.Length; i++)    // 处理数组
         {
            inVals[i] = inVals[i] * 10;
            Console.WriteLine("{0}  ", inVals[i]); // 显示新值
         }
   }
}

class Program
{
   static void Main()
   {
      int first = 5, second = 6, third = 7;        // 声明3个int

      MyClass mc = new MyClass();
      mc.ListInts( first, second, third );         // 调用方法
                              ↑
                             实参
      Console.WriteLine("{0}, {1}, {2}", first, second, third);
   }
}
```

这段代码产生如下输出：

```
50
60
70
5, 6, 7
```

图5-10阐明了在方法执行的不同阶段实参和形参的值：

- 方法调用之前，三个实参已经在栈里。
- 在方法的开始，三个实参被用于初始化堆中的数组，并且数组的引用被赋值给形参inVals。
- 在方法内部，代码首先检查以确认数组引用不是null，然后处理数组，把每个元素乘以10并保存回去。
- 方法执行之后，形参inVals失效。

关于参数数组，需要记住的重要一点是当数组在堆中被创建时，实参的值被复制到数组中。在这方面，它们像值参数。

- 如果数组参数是值类型，那么值被复制，实参不受方法内部的影响。
- 如果数组参数是引用类型，那么引用被复制，实参引用的对象可以受到方法内部的影响。

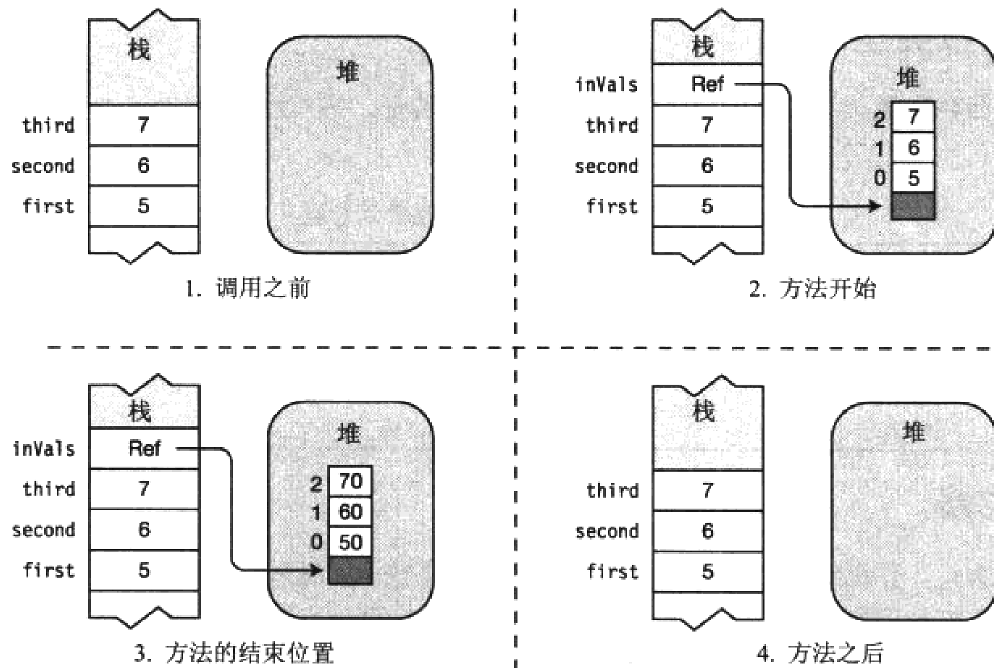



图5-10 参数数组

## 5.10.2 数组作实参

也可以在方法调用之前创建并组装一个数组，把单一的数组变量作为实参传递。这种情况下，编译器使用你的数组而不是重新创建一个。

例如，下面代码使用前一个示例中声明的方法ListInts。在这段代码中，Main创建一个数组，并用数组变量而不是使用分离的整数作为实参。

```
static void Main()
{
   int[] MyArr = new int[] { 5, 6, 7 };  // 创建并实例化数组

   MyClass mc = new MyClass();
   mc.ListInts(MyArr);                   // 调用方法

   foreach (int x in MyArr)
      Console.WriteLine("{0}", x);       // 输出每个元素
}
```

这段代码产生以下输出：

```
50
60
70
```

## 5.11 参数类型总结

因为有四种参数类型，有时很难记住它们的不同特征。表5-2对它们做了总结，使之更易于比较和对照。

表5-2 参数类型语法使用总结

| 参数类型 | 修饰符 | 是否在声明时使用 | 是否在调用时使用 | 执 行 |
|---|---|---|---|---|
| 值 | 无 | | | 系统把实参的值复制到形参 |
| 引用 | ref | 是 | 是 | 形参是实参的别名 |
| 输出 | out | 是 | 是 | 形参是实参的别名 |
| 数组 | params | 是 | 否 | 允许传递可变数目的实参到方法 |

## 5.12 栈帧

到现在为止，你已经知道本地变量和参数被保存在栈里。让我们更深入一点看看这个结构。

当一个方法被调用时，在栈顶分配了一块内存用于保存一定数量与方法相关的数据项，这块内存叫方法的栈帧（stack frame）。

- 栈帧含有保存下列内容的内存：
  - 返回地址——就是方法退出时在哪儿继续执行。
  - 分配内存的参数——就是方法的值参数，如果有参数数组以及参数数组。
  - 与方法调用相关的其他各种管理数据项。
- 方法被调用时，它的整个栈帧被压入栈中。
- 方法退出时，它的整个栈帧被从栈中弹出。弹出一个栈帧有时也称为释放栈。

例如，下面的代码声明了三个方法。Main调用MethodA，MethodA调用MethodB，共创建了三个栈帧。当方法退出时，栈被释放了。

```
class Program
{
   static void MethodA( int par1, int par2)
   {
      Console.WriteLine("Enter MethodA: {0}, {1}", par1, par2);
      MethodB(11, 18);                                   // 调用MethodB.
      Console.WriteLine("Exit  MethodA");
   }

   static void MethodB(int par1, int par2)
   {
      Console.WriteLine("Enter MethodB: {0}, {1}", par1, par2);
      Console.WriteLine("Exit  MethodB");
   }

   static void Main( )
```

```
    {
        Console.WriteLine("Enter Main");
        MethodA( 15, 30);                                    // 调用MethodA
        Console.WriteLine("Exit  Main");
    }
}
```

这段代码产生如下输出：

```
Enter Main
Enter MethodA: 15, 30
Enter MethodB: 11, 18
Exit  MethodB
Exit  MethodA
Exit  Main
```

图5-11展示了方法调用时每个方法的栈帧是如何放到栈里的，以及方法完成时栈是如何被释放的。

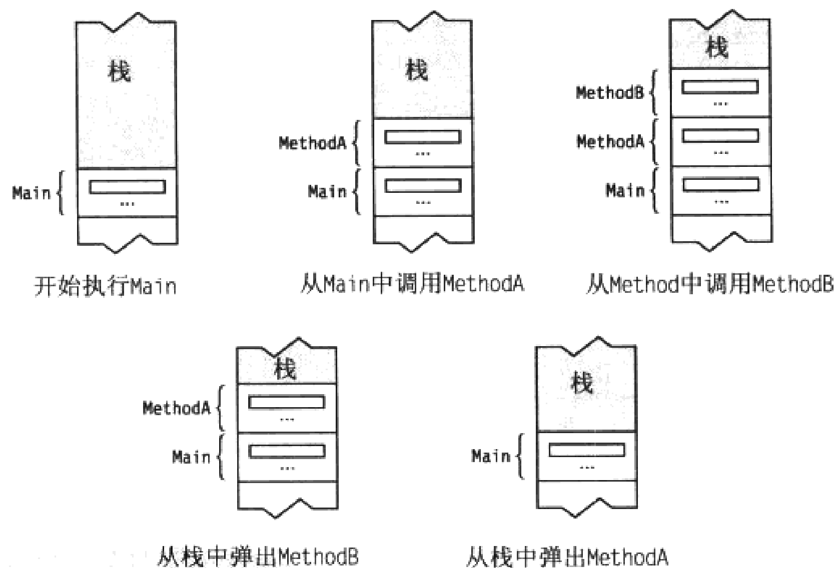



图5-11 一个简单程序中的栈帧

## 5.13 递归

除了调用其他方法，方法还能调用自己，这称为递归（recursion）。

递归可以产生一些非常巧妙的代码，比如下面的计算数字阶乘的方法。注意方法的内部，方法调用自己用的实参比自己的输入参数少1。

```
int Factorial(int inValue)
{
   if (inValue <= 1)
      return inValue;
```

```
   else
      return inValue * Factorial(inValue - 1);      // 再次调用Factorial
                                ↑
                              调用自己
}
```

方法调用自己的机制和调用其他方法完全相同。方法的每次调用都会有一个新的帧被压入栈中。

例如，在下面的代码中，方法Count使用比它的输入参数少1的参数调用自己，然后打印出它的输入参数。当递归越来越深时，栈也越来越大。

```
class Program
{
   public void Count(int inVal)
   {
      if (inVal == 0)
         return;
      Count(inVal - 1);                 // 再次调用方法
           ↑
         调用自己

      Console.WriteLine("{0} ", inVal);
   }

   static void Main()
   {
      Program pr = new Program();
      pr.Count(3);
   }
}
```

这段代码产生以下输出：

```
1
2
3
```

图5-12阐明了这段代码。请注意，输出值为3时，方法Count有四个不同的、独立的栈帧，每个帧的输入参数inVal都有自己的值。

## 5.14　方法重载

一个类中可以有一个以上的方法拥有相同的名称，这叫做**方法重载**（method overload）[①]。使用相同名称的每个方法必须有一个和其他方法不相同的**签名**（signature）。

- 方法的签名由下列信息组成，它们在方法声明的方法头中：
  - 方法的名称；
  - 参数的数目；

① 请注意此概念与继承中“方法覆写”（method override）的不同，参见7.5.1节。——编者注

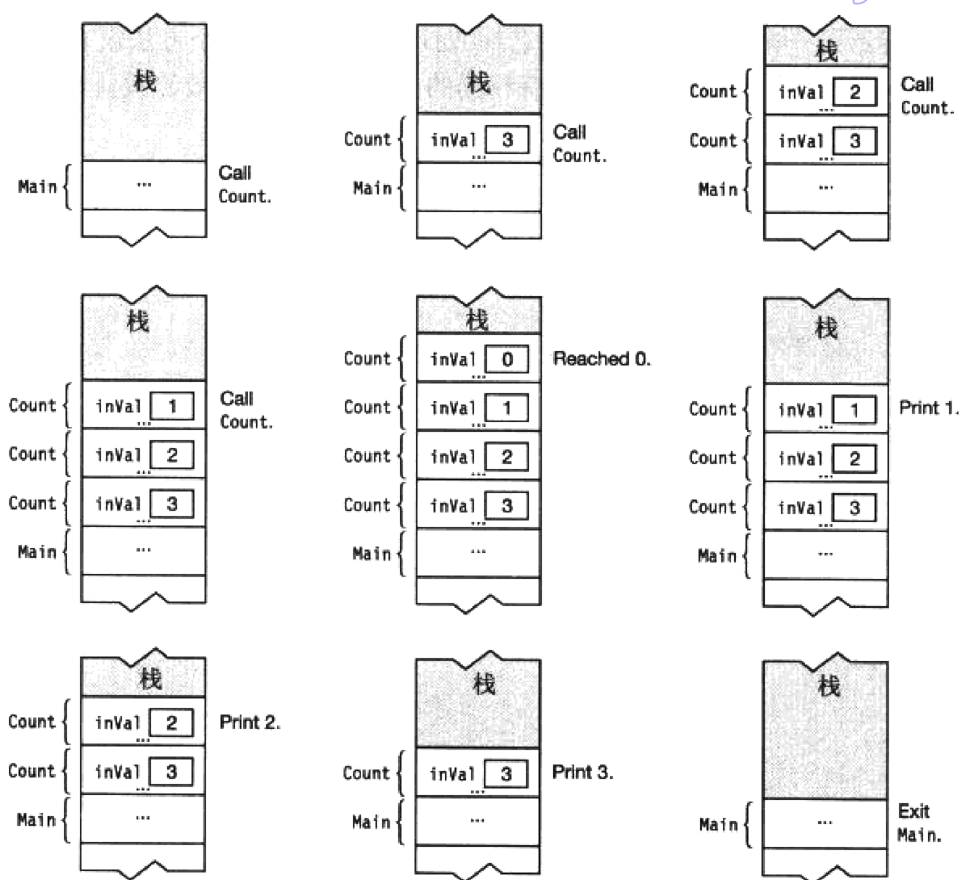



图5-12 递归示例

- 参数的数据类型和顺序；
- 参数修饰符。

❑ 返回类型不是签名的一部分，而认为它是签名的一部分是一种常见错误。

❑ 形参的名称也不是签名的一部分。

不是签名的一部分

```
long AddValues( int a, out int b) { ... }
```

签名

例如，下面四个方法是方法名AddValues的重载：

```
class A
{
   long AddValues( int   a, int   b)         { return a + b; }
   long AddValues( int   a, int   b, int c)  { return a + b + c; }
   long AddValues( float a, float b)         { return a + b; }
   long AddValues( long  a, long  b)         { return a + b; }
}
```

下面的代码展示了一个非法的重载方法名AddValues的企图。两个方法仅有返回类型和形参名不同，但它们仍有相同的签名，因为它们有相同的方法名，而且参数的数目、类型和顺序也相同。编译器会对这段代码生成一条错误信息。

```
class B                 签名
{                        ↓
   long AddValues( long  a, long  b) { return a+b; }
   int  AddValues( long  c, long  d) { return c+d; }
}                        ↑
                        签名
```

# 第6章 类 进 阶

**本章内容**

- ❑ 类成员
- ❑ 实例类成员
- ❑ 静态字段
- ❑ 从类的外部访问静态成员
- ❑ 静态函数成员
- ❑ 其他静态类成员类型
- ❑ 成员常量
- ❑ 属性
- ❑ 实例构造函数
- ❑ 静态构造函数
- ❑ 对象初始化列表
- ❑ 析构函数
- ❑ 比较构造函数和析构函数
- ❑ readonly修饰符
- ❑ this关键字
- ❑ 索引
- ❑ 访问器的访问修饰符
- ❑ 分部类和分部类型

## 6.1 类成员

前两章阐述了九种类成员类型中的两种：字段和方法。在这一章中，我会介绍更多类型的类成员，并讨论类成员的生命周期。

表6-1列出了类的成员类型。已经介绍过的类型用菱形标记。将在本章阐述的类型用勾号标记。将在以后的文字中阐述的类型用空的选择框标记。

表6-1　类成员的类型

| 数据成员（保存数据） | 函数成员（执行代码） | |
|---|---|---|
| ◆字段 | ◆方法 | ✔运算符 |
| ✔常量 | ✔属性 | ✔索引 |
| | ✔构造函数 | ❑事件 |
| | ✔Finalizer方法 | |

## 成员修饰符的顺序

在前面的内容中，你看到字段和方法的的声明可以包括如public和private这样的修饰符。在这一章中，会讨论许多其他的修饰符。因为多个修饰符可以在一起使用，出现的问题就是：它们需要按什么顺序排列呢？

类成员声明语句由下列部分组成：核心声明、一组可选的修饰符（modifier）和一组可选的特性（attribute）。用于描述这个结构的语法如下。方括号表示方括号内的成分是可选的。

```
[ 特性 ]   [ 修饰符 ]   核心声音
```

- ❑ 如果有修饰符，它必须放在核心声明之前。
- ❑ 如果有特性，它必须放在修饰符和核心声明之前。

如果一个声明有多个修饰符，它们可以以任何顺序放在核心声明之前。到现在为止，只讨论了两个修饰符：public和private。如果有多个特性，它们可以以任何顺序放在修饰符之前。我会在第24章阐述特性。

例如，public和static都是修饰符，可以用在一起修饰某个声明。因为它们都是修饰符，所以可以放置成任何顺序。下面两行代码是语义等价的：

```
public static int MaxVal;

static public int MaxVal;
```

图6-1阐明了声明中各成分的顺序，它们应用到两种成员类型：字段和方法。注意，字段的类型和方法的返回类型不是修饰符——它们是核心声明的一部分。

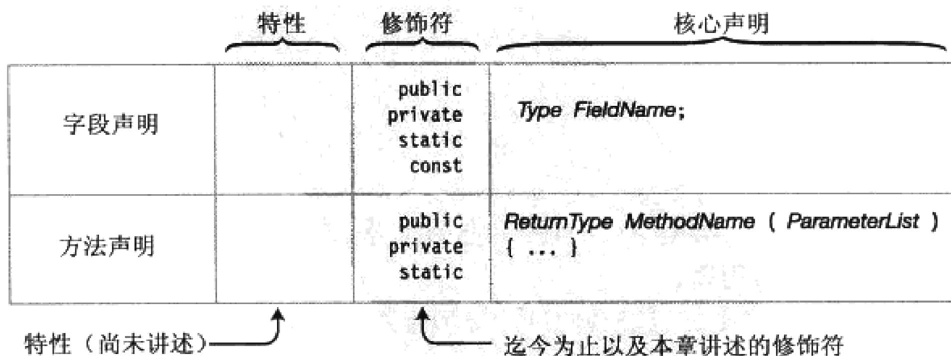



图6-1　特性、修饰符和核心声明的顺序

## 6.2 实例类成员

类成员可以关联到一个实例，也可以关联到类的整体。默认情况下，成员被关联到一个实例。可以想象类的每个实例拥有自己的各个类成员的拷贝，这些成员称为实例成员。

改变一个实例字段的值不会影响任何其他实例中成员的值。迄今为止，你所看到的字段和方法都是实例字段和实例方法。

例如，下面的代码声明了一个类D，它带有唯一整型字段Mem1。Main创建了两个该类的实例，每个实例都有自己的字段Mem1的拷贝，改变一个实例的字段拷贝的值不影响其他实例的拷贝的值。图6-2阐明了类D的两个实例。

```
class D
{
   public int Mem1;
}

class Program
{
   static void Main()
   {
      D d1 = new D();
      D d2 = new D();
      d1.Mem1 = 10; d2.Mem1 = 28;

      Console.WriteLine("d1 = {0}, d2 = {1}", d1.Mem1, d2.Mem1);
   }
}
```

这段代码产生如下输出：

```
d1 = 10, d2 = 28
```

类D的每个实例都有自己的字段Mem1的拷贝

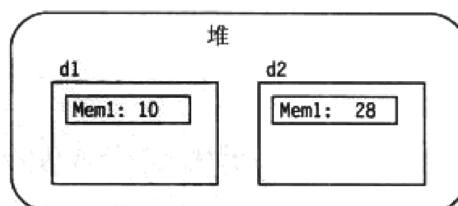



图6-2 两个有实例数据成员的实例

## 6.3 静态字段

除了实例字段，类还可以拥有静态字段。

- 静态字段被类的所有实例共享，所有实例都访问同一内存位置。因此，如果该内存位置的值被一个实例改变了，这种改变对所有的实例都可见。

❑ 使用static修饰符声明一个字段为静态，如：

```
class D
{
   int Mem1;                    // 实例字段
   static int Mem2;             // 静态字段
     ↑
}  关键字
```

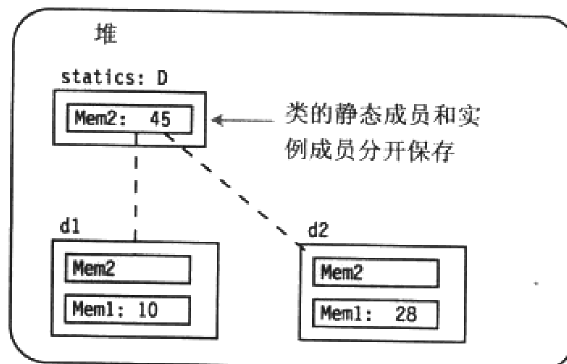
例如，图6-3左边的代码声明了类D，它含有静态字段Mem2和实例字段Mem1。Main定义了两个类D的实例。该图表明静态成员Mem2是与所有实例的存储分开保存的。实例中灰色的字段表明，从实例内部，静态成员看上去和任何其他成员字段一样。

❑ 因为Mem2是静态的，类D的两个实例共享单一的Mem2字段。如果Mem2在一个实例中被改变，它在另一个中也被改变了。

❑ 成员Mem1没有声明为static，所以每个实例都有自己的拷贝。

```
class D
{
   int Mem1;
   static int Mem2;
   ...
}

static void Main()
{
   D d1 = new D();
   D d2 = new D();
   ...
}
```




静态成员Mem2被类D的所有实例共享，而每个实例都有自己的实例字段Mem1的拷贝

图6-3 静态和非静态数据成员

## 6.4 从类的外部访问静态成员

在前一章中，你看到点运算符被用于从类的外部访问实例的成员。点运算符由实例名、点、成员名组成。

就像实例成员，静态成员也可以使用点运算符从类的外部访问。但因为没有实例，所以必须使用类名，如下面代码所示：

```
类名
↓
D.Mem2 = 5;            // 访问静态类成员
   ↑
成员名
```

## 6.4.1　静态字段示例

下面的代码扩展了前文的类D，增加两个方法：

- 一个方法设置两个数据成员的值。
- 另一个方法显示两个成员变量的值。

```
class D {
   int        Mem1;
   static int Mem2;

   public void SetVars(int v1, int v2) // 设置值
   {  Mem1 = v1; Mem2 = v2; }
                  ↑ 像访问实例字段一样访问它

   public void Display( string str )
   {  Console.WriteLine("{0}: Mem1= {1}, Mem2= {2}", str, Mem1, Mem2); }
}                                                          ↑
                                                  像访问实例字段一样访问它
class Program {
   static void Main()
   {
      D d1 = new D(), d2 = new D();   // 创建两个实例

      d1.SetVars(2, 4);               // 设置d1的值
      d1.Display("d1");

      d2.SetVars(15, 17);             // 设置d2的值
      d2.Display("d2");

      d1.Display("d1");          // 再次显示d1,
   }                             // 这时Mem2静态成员的值已改变
}
```

这段代码产生以下输出：

```
d1: Mem1= 2, Mem2= 4
d2: Mem1= 15, Mem2= 17
d1: Mem1= 2, Mem2= 17
```

## 6.4.2　静态成员的生存期

实例成员在实例被创建时开始存在，当实例被销毁时停止存在。然而，静态成员即使没有类的实例也存在并可以访问。

图6-4阐述了类D，它带有一个静态字段Mem2。虽然Main没有定义类的任何实例，但它把值5赋给该静态字段并把它打印出来。

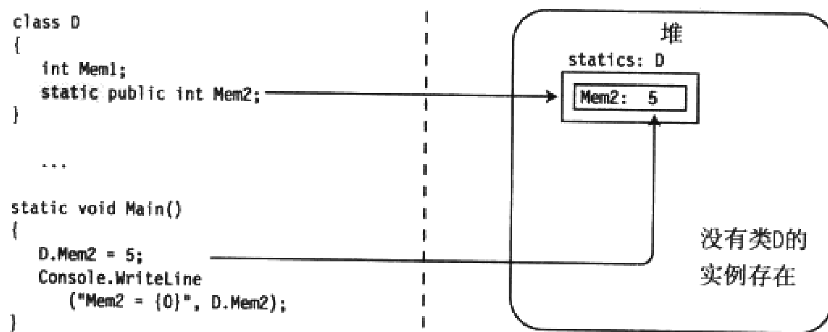



图6-4　无类实例的静态成员

图6-4中的代码产生以下输出：

```
Mem2 = 5
```

**说明**　静态成员即使没有类的实例也存在。如果静态字段有初始化语句，那么该字段在类的任何静态成员被使用之前初始化，但没必要在程序执行的开始就初始化。

## 6.5　静态函数成员

除了静态字段，还有静态函数成员。

- 如同静态字段，静态函数成员独立于任何类实例。即使没有类的实例，仍然可以调用静态方法。
- 静态函数成员不能访问实例成员。然而，它们能访问其他静态成员。

例如，下面的类包含一个静态字段和一个静态方法。注意，静态方法的方法体访问静态字段。

```
class X
{
    static public int A;                          // 静态字段
    static public void PrintValA()                // 静态方法
    {
       Console.WriteLine("Value of A: {0}", A);
    }                                        ↑
}                                       访问静态字段
```

下面的代码使用前文代码中定义的类X。图6-5阐释了这段代码。

```
class Program
{
   static void Main()
   {
      X.A = 10;                 // 使用点号语法
      X.PrintValA();            // 使用点号语法
   } ↑
}   类名称
```

这段代码产生以下输出：

```
Value of A: 10
```

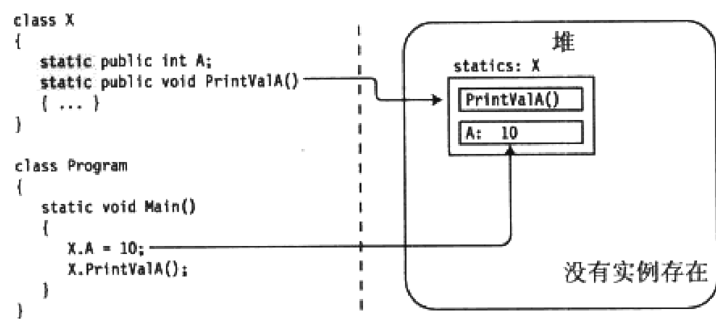



图6-5 即使没有类的实例，类的静态方法也可以被调用

## 6.6 其他静态类成员类型

可以声明为static的类成员类型在表6-2中被选中。其他成员类型不能声明为static。

表6-2 可以声明为静态的类成员类型

| 数据成员（储存数据） | 函数成员（执行代码） |
|---|---|
| ✔ 字段 | ✔ 方法 |
| 常量 | ✔ 属性 |
| | ✔ 构造函数 |
| | ✔ 运算符 |
| | 索引 |
| | ✔ 事件 |

**说明** *与C和C++不同，在C#中没有全局常量。每个常量都必须声明在类型内。*

## 6.7 成员常量

成员常量就像前一章所述的本地常量，只是它们被声明在类声明中，如下面的示例：

```
class MyClass
{
    const int IntVal = 100;       // 定义int类型常量
          ↑           ↑           // 值为100
         类型        初始值
}

const double PI = 3.1416;         // 错误：不能在类型声明
                                  // 之外声明
```

就像本地常量，用于初始化成员常量的值在编译期必须是可计算的，而且通常是一个预定义

简单类型或由它们组成的表达式。

```
class MyClass
{
   const int IntVal1 = 100;
   const int IntVal2 = 2 * IntVal1;  // 没问题，IntVal1的值
}                                    // 前面一行已设置
```

就像本地常量，不能在成员常量声明以后给它赋值。

```
class MyClass
{
   const int IntVal;                 // 错误：必须初始化
   IntVal = 100;                     // 错误：不允许赋值
}
```

## 常量就像静态量

然而，成员常量比本地常量更有趣，它们表现得像静态变量。它们对类的每个实例都是“可见的”，而且即使没有类的实例它们也可以使用。

例如，下面的代码声明了类X，带有常量字段PI。Main没有创建X的任何实例，但仍然可以使用字段PI并打印它的值。

```
class X
{
   public const double PI = 3.1416;
}

class Program
{
   static void Main()
   {
      Console.WriteLine("pi = {0}", X.PI);    // 使用静态字段
   }
}
```

这段代码产生以下输出：

```
pi = 3.1416
```

然而，与真正的静态量不同，常量没有自己的存储位置，而是在编译时被编译器替换。这种方式类似于C和C++中的#define值。如图6-6所示，它阐述了前面的代码。因此，虽然常量成员表现像一个静态量，但不能声明一个常量为static。

```
static const double PI = 3.14;

Error: can't declare a constant as static
```

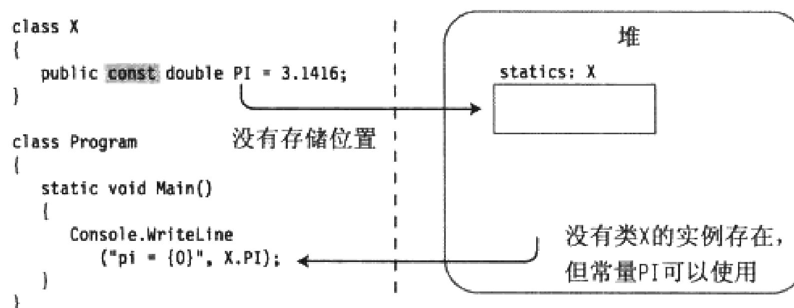



图6-6 常量字段表现像静态字段，但是在内存中没有存储位置

## 6.8 属性

属性是代表类的实例或类中的一个数据项的成员。

使用属性看起来非常像写入或读取一个字段，语法是相同的。例如，下面的代码展示了名称为MyClass的类的使用，它有一个公有字段和一个公有属性。从使用上无法区分它们。

```
MyClass mc = new MyClass();

mc.MyField    = 5;                          // 给字段赋值
mc.MyProperty = 10;                         // 给属性赋值

WriteLine("{0} {1}", mc.MyField, mc.MyProperty); // 读取字段和属性
```

就像字段，属性有如下特征：

- 它是命名的类成员。
- 它有类型。
- 它可以被赋值和读取。

然而和字段不同，属性是一个函数成员。

- 它不为数据存储分配内存！
- 它执行代码。

属性是指定的一组两个匹配的、称为访问器的方法。

- set访问器用于为属性赋值。
- get访问器用于从属性获取值。

图6-7展示了属性的表示法。左边的代码展示了声明一个名称为MyValue的int型属性的语法，

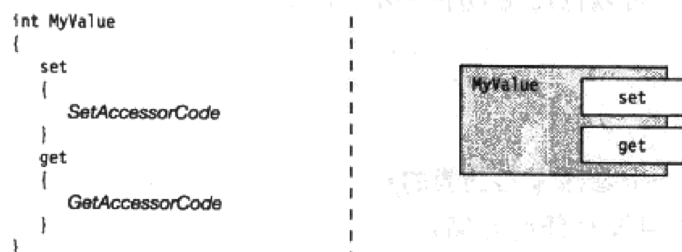



图6-7 int型、名称为MyValuer的属性示例

右边的图像展示了属性在本文中将被如何可视化地显示。请注意，访问器被显示为从后面伸出，因为它们不能被直接调用。这一点你很快会看到。

### 6.8.1　属性声明和访问器

set和get访问器有预定义的语法和语义。可以把set访问器想象成一个方法，带有单一的参数“设置”属性的值。get访问器没有参数并从属性返回一个值。

- set访问器总是：
  - 一个单独的、隐式的值参，名称为value，与属性的类型相同。
  - 一个返回类型void。
- get访问器总是：
  - 没有参数。
  - 一个与属性类型相同的返回类型。

属性声明的结构如图6-8所示。注意，图中的访问器声明既没有显式的参数，也没有返回类型声明。不需要它们，因为它们已经在属性的类型中隐含了。

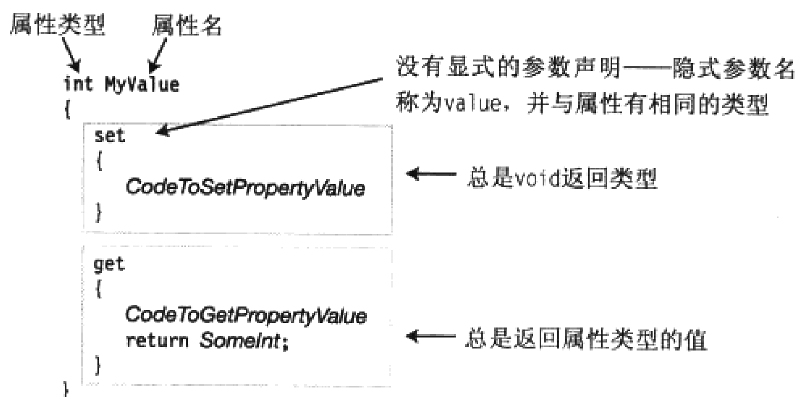



图6-8　属性声明的语法结构

set访问器中的隐式参数value是一个普通的值参。和其他值参一样，可以用它发送数据到方法体或这种情况中的访问器块。在块的内部，可以像普通变量那样使用value，包括对它赋值。

访问器的其他重点有：

- get访问器的所有执行路径必须包含一条return语句，返回一个属性类型的值。
- 访问器set和get可以以任何顺序声明，并且，除了这两个访问器外在属性上不允许有其他方法。

### 6.8.2　属性示例

下面的代码展示了一个名称为C1的类的声明示例，它含有一个名称为MyValue的属性。

- 请注意，属性本身没有任何存储。取而代之，访问器决定如何处理发进来的数据，以及什么数据应被发送出去。在这种情况下，属性使用一个名称为TheRealValue的字段

作存储。

- set访问器接受它的输入参数value，并把它的值赋给字段TheRealValue。
- get访问器只是返回字段TheRealValue的值。

图6-9说明了这段代码。

```
class C1
{
   private int TheRealValue;              // 字段：内存分配

   public int MyValue                     // 属性：未分配内存
   {
      set
      {
         TheRealValue = value;
      }

      get
      {
         return TheRealValue;
      }
   }
}
```

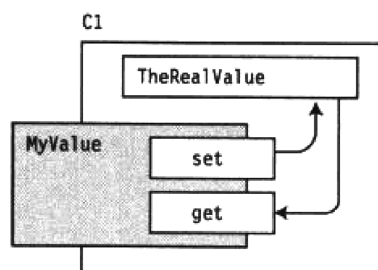



图6-9 属性访问器常常使用字段作存储

### 6.8.3 使用属性

写入和读取属性的方法与访问字段一样。访问器被隐式调用。

- 要写入一个属性，在赋值语句的左边使用属性的名称。
- 要读取一个属性，把属性的名称用在表达式中。

例如，下面的代码包含一个名称为MyValue的属性声明的轮廓。只使用属性名写入和读取属性，如同它是一个字段名。

```
int MyValue              // 属性声明
{
   set{ ... }
   get{ ... }
}
```

```
...
属性名称
  ↓
MyValue = 5;               // 赋值：隐式调用set方法
z = MyValue;               // 表达式：隐式调用get方法
     ↑
   属性名称
```

依据写入或读取属性，适当的访问器会被隐式调用，不能显式地调用访问器。试图这样做会产生一个编译错误。

```
y = MyValue.get();        // 错误！不能显式调用访问器
MyValue.set(5);           // 错误！不能显式调用设置器
```

### 6.8.4　属性和关联字段

属性常和字段关联，我们在前两节已经看到了。惯例是在类中将字段声明为private以封装一个字段，并声明一个public属性以提供受控的从类外部对字段的访问。和属性关联的字段常被称为*后备字段*或*后备存储*。

例如，下面的代码使用公有属性MyValue以提供对私有字段TheRealValue的受控访问。

```
class C1
{
   private int TheRealValue = 10;   // 后备字段：分配内存
   public  int MyValue              // 属性：不分配内存
   {
      set{ TheRealValue = value; }  // 设置TheRealValue字段的值
      get{ return TheRealValue; }   // 获取字段的值
   }
}

class Program
{
   static void Main()
   {
                               把属性看作一个字段，从中读取它的值
      C1 c = new C1();                           ↓
      Console.WriteLine("MyValue:  {0}", c.MyValue);

      c.MyValue = 20;      ← 使用赋值语句设置属性的值
      Console.WriteLine("MyValue:  {0}", c.MyValue);
   }
}
```

属性和它们的后备字段有几种命名约定。一种约定是两个名称使用相同的内容，但字段使用Camel大小写（首字母小写），属性使用Pascal大小写。虽然这违反了“仅使用大小写区分不同标识符是个坏习惯”这条普遍规则，但它有个好处，可以把两个标识符以一种有意义的方式联系在一起。

另一种约定是属性使用Pascal大小写，对于字段，使用Camel大小写版的相同标识符，并以下

划线开始。

下面的代码展示了两种约定：

```
private int firstField;                    // Camel大小写
public  int FirstField                     // Pascal大小写
{
   get { return firstField; }
   set { firstField = value; }
}

private int _secondField;                  // 下划线and camel大小写
public  int SecondField
{
   get { return _secondField; }
   set { _secondField = value; }
}
```

## 6.8.5 执行其他计算

属性访问器并不局限于仅仅对关联的后备字段传进传出数据。访问器get和set能执行任何计算，或不执行任何计算。唯一必需的行为是get访问器要返回一个属性类型的值。

例如，下面的示例展示了一个有效的（但可能没有用处）属性，仅在get访问器被调用时返回值5。当set访问器被调用时，它不做任何事情。隐式参数value的值被忽略了。

```
public int Useless
{
   set{  /* I'm not setting anything.           */ }
   get{  /* I'm just returning the value 5.    */
      return 5;
   }
}
```

下面的代码展示一个更现实和有用的属性，set访问器在设置关联字段之前实现过滤。set访问器把字段TheRealValue的值设置成输入值——除非输入值大于100，在这种情况中，它设置TheRealValue为100。

```
int TheRealValue = 10;                        // 字段
int MyValue                                   // 属性
{
   set {                                      // 设置字段值
      TheRealValue = value > 100              // 确保值不大于100
                        ? 100
                        : value;
   }
   get {                                      // 获取字段的值
      return TheRealValue;
   }
}
```

**说明**　在上面的代码示例中，从等号到语句结尾部分的语法可能看起来有些奇怪。该表达式使用条件运算符，在第8章将会非常详细地阐述。条件运算符是一种三元运算符，计算问号之前的表达式，如果表达式计算结果为true，那么返回问号后的第一个表达式，否则，它返回冒号之后的表达式。

## 6.8.6　只读和只写属性

可以通过忽略访问器的声明，以使一个或其他的属性访问器不被定义。

- 只有get访问器的属性称为只读属性。只读属性是一种安全的方法，把一项数据从类或类的实例中传出，而不允许太多的访问。
- 只有set访问器的属性称为只写属性。只写属性是把一项数据从类的外部传入类而不允许太多访问的安全方法。
- 两个访问器中至少有一个必须定义，否则编译器会产生一条错误信息。

图6-10阐述了只读和只写属性。




图6-10　属性可以有一个或其他的访问器不被定义

## 6.8.7　计算只读属性示例

到迄今为止，在大多示例中，属性都和一个字段关联，并且get和set访问器引用该字段。然而，属性并非必须和字段关联。在下面的示例中，get访问器计算返回值。

在下面的示例代码中，类RightTriangle表示一个直角三角形，毫不惊奇。

- 它有两个公有字段，表示直角三角形的两个直角边的长度。这些字段可以被写入和读取。
- 第三边由属性Hypotenuse表示，它是一个只读属性，它的返回值基于另外两个边的长度。它没有储存在字段中。相反，它在需要时根据当前A和B的值计算正确的值。

图6-11阐释了只读属性Hypotenuse。

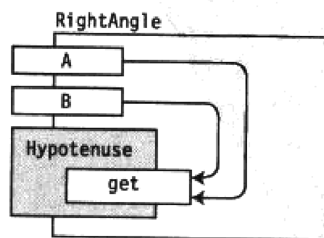



图6-11　只读属性Hypotenuse

```
class RightTriangle
{
   public double A = 3;
   public double B = 4;
   public double Hypotenuse                    // 只读属性
   {
      get{ return Math.Sqrt((A*A)+(B*B)); }    // 计算返回值
   }
}

class Program
{
   static void Main()
   {
      RightTriangle c = new RightTriangle();
      Console.WriteLine("Hypotenuse:  {0}", c.Hypotenuse);
   }
}
```

### 6.8.8 属性和数据库示例

另一个属性不与字段关联的例子是属性和数据库中的值关联。在这种情况中，get访问器进行适当的数据库调用以从数据库中获取值。set访问器进行相应的数据库调用以把新的值设置到数据库中。

例如，下面的属性被关联到某数据库中的一个特定值。这段代码假定类中有另外两个方法处理数据库事务的细节：

- SetValueInDatabase接受一个整型参数，并用它设置某数据库的记录中的一个特定的字段。
- GetValueFromDatabase从某数据库的特定记录中获取并返回一个特定的整型字段值。

```
int MyDatabaseValue
{
   set                          // 在数据库中设置整型值
   {
      SetValueInDatabase(value);
   }
   get                          // 从数据库中获取整型值
   {
      return GetValueFromDatabase();
   }
}
```

### 6.8.9 自动实现属性

因为属性经常被关联到后备字段，C# 3.0增加了自动实现属性（automatically implemented property或auto-implemented property），允许只声明属性而不声明后备字段。

自动实现属性的要点如下：

- ❑ 不声明后备字段——编译器根据属性的类型分配存储。
- ❑ 不能提供访问器的方法体——它们被简单地声明为分号。get担当简单的内存读，set担当简单的写。
- ❑ 除非通过访问器，否则不能访问后备字段。因为不能用其他的方法访问它，所以只读和只写自动实现属性也就没有意义，因此它们不被允许。

下面的代码展示了一个自动实现属性的示例。

```
class C1
{                ← 没有声明后备字段
   public int MyValue                           // 分配内存
   {
      set; get;
   }    ↑    ↑
}      访问器的方法体被声明为分号

class Program
{
   static void Main()
   {                            像使用规则属性那样使用自动属性
      C1 c = new C1();                    ↓
      Console.WriteLine("MyValue:  {0}", c.MyValue);

      c.MyValue = 20;
      Console.WriteLine("MyValue:  {0}", c.MyValue);
   }
}
```

这段代码产生以下输出：

```
MyValue:  0
MyValue:  20
```

除了方便以外，自动实现属性使你在倾向于声明一个公有字段的地方容易地插入一个属性。公有字段通常是不推荐的，因为它们不允许像使用访问器那样处理输入和输出。

然而，你可能倾向于发布一个带有公有字段的代码版本，然后在下一版把字段改成属性。但是，已编译的变量和已编译的属性在语义上是不同的。如果在后来的版本中，你把字段切换成属性，那么任何访问第一个版中该字段的程序集都必需重新编译以使用属性。如果在第一次就使用了属性，那么使用者就不必重新编译了。

### 6.8.10　静态属性

属性也可以声明为static。静态属性的访问器和所有静态成员一样：

- ❑ 不能访问类的实例成员——虽然它们能被实例成员访问。
- ❑ 存在，不管类是否有实例。
- ❑ 当从类的外部访问时，必需使用类名引用，而不是实例名。

例如，下面的代码展示了一个类，它带有名称为MyValue的静态属性，并关联到一个名称为myValue的静态字段。在Main的开始三行，即使没有类的实例，属性也被访问了。Main的最后一行调用一个实例方法，它从类的内部访问属性。

```
class Trivial
{
   static int myValue;
   public static int MyValue
   {
      set { myValue = value; }
      get { return myValue;  }
   }

   public void PrintValue()                     从类的内部访问
   {                                                ↓
      Console.WriteLine("Value from inside: {0}", MyValue);
   }
}

class Program
{
   static void Main()                          从类的外部访问
   {                                               ↓
      Console.WriteLine("Init Value: {0}", Trivial.MyValue);
      Trivial.MyValue = 10;          ← 从类的外部访问
      Console.WriteLine("New Value : {0}", Trivial.MyValue);

      Trivial tr = new Trivial();
      tr.PrintValue();
   }
}
```

```
Init Value: 0
New Value : 10
Value from inside: 10
```

## 6.9 实例构造函数

实例构造函数[①]是一个特殊的方法，它在类的每个新实例创建的时候执行。

- 构造函数用于初始化类实例的状态。
- 如果希望能从类的外部创建类的实例，需要声明构造函数为public。

图6-12阐述了构造函数的语法。构造函数看起来很像类声明中的其他方法，除了下面这几点：

- 构造函数的名称和类名相同。
- 构造函数不能有返回值。

① “constructor”一词微软官方文档中也常译为“构造器”。——编者注

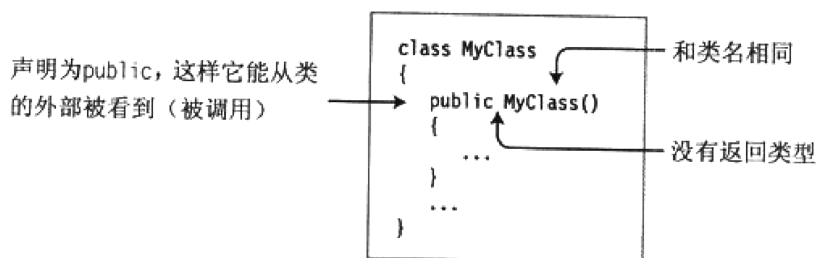



图6-12　构造函数声明

例如，下面的类使用构造函数初始化它的字段。本例中，它有一个名称为TimeOfInstantiation的字段被初始化为当前的日期和时间。

```
class MyClass
{
    DateTime TimeOfInstantiation;                        // 字段
    ...
    public MyClass()                                     // 构造函数
    {
        TimeOfInstantiation = DateTime.Now;              // 初始化字段
    }
    ...
}
```

**说明**　已经结束了静态属性这一节，仔细看看初始化TimeOfInstantiation那一行。DateTime类是从BCL中引入的，Now是类DateTime的静态属性。Now属性创建了一个新的DateTime类的实例，初始化它为系统时钟中的当前日期和时间，并返回新DateTime实例的引用。

### 6.9.1　带参数的构造函数

构造函数在下列方面和其他方法相似：

- 构造函数可以带参数。参数的语法和其他方法完全相同。
- 构造函数可以被重载。

在使用创建对象表达式创建类的新实例时，要提供new运算符，并跟着类的构造函数之一。new运算符使用该构造函数创建类的实例。

例如，在下面的代码中，Class1有三个构造函数：一个不带参数、一个带int参数以及另一个带string参数。Main使用每一个构造函数创建实例。

```
class Class1
{
    int Id;
    string Name;

    public Class1()           { Id=28;    Name="Nemo"; }   // 构造函数 0
    public Class1(int val)    { Id=val;   Name="Nemo"; }   // 构造函数 1
```

```
    public Class1(String name) { Name=name;            }  // 构造函数 2

    public void SoundOff()
    { Console.WriteLine("Name {0},   Id {1}", Name, Id); }
}

class Program
{
    static void Main()
    {
        Class1 a = new Class1(),                  // 调用构造函数 0
               b = new Class1(7),                 // 调用构造函数 1
               c = new Class1("Bill");            // 调用构造函数 2

        a.SoundOff();
        b.SoundOff();
        c.SoundOff();
    }
}
```

这段代码产生以下输出：

```
Name Nemo,   Id 28
Name Nemo,   Id 7
Name Bill,   Id 0
```

## 6.9.2 默认构造函数

如果在类的声明中没有显式地提供实例构造函数，那么编译器会提供一个隐式的默认构造函数，它有以下特征：

- 它不带参数。
- 它的方法体为空。

如果程序员定义了一个或多个构造函数，那么编译器将不会为该类定义默认构造函数。

例如，Class2声明了两个构造函数。

- 因为已经至少有一个显式定义的构造函数，编译器不会创建任何附加构造函数。
- 在Main中，试图使用不带参数的构造函数创建新的实例。因为没有零参数的构造函数，所以编译器会产生一条错误信息。

```
class Class2
{
    public Class2(int Value)    { ... }   // 构造函数 0
    public Class2(String Value) { ... }   // 构造函数 1
}

class Program
{
```

```
    static void Main()
    {
       Class2 a = new Class2();   // 错误！没有不带参数的构造函数
       ...
    }
}
```

## 6.10 静态构造函数

构造函数也可以声明为static。实例构造函数初始化类的每个新实例，而static构造函数初始化类层次的项目。通常，静态构造函数初始化类的静态字段。

- ❑ 类层次的项目需要被初始化：
  - ■ 在任何静态成员被引用之前。
  - ■ 在类的任何实例被创建之前。
- ❑ 静态构造函数在下列方面就像实例构造函数：
  - ■ 静态构造函数的名称必须和类名相同。
  - ■ 构造函数不能返回值。
- ❑ 静态构造函数在下列方面和实例构造函数不同：
  - ■ 静态构造函数声明中使用static关键字。
  - ■ 类只能有一个静态构造函数，而且不能带参数。
  - ■ 静态构造函数不能有访问修饰符。

下面是一个静态构造函数的示例。注意它的形式和实例构造函数相同，只是增加了static关键字。

```
class Class1
{
   static Class1 ()
   {
      ...                  // 静态构造函数的函数体
   }
   ...
```

关于静态构造函数应该了解的其他重要内容如下：

- ❑ 类既可以有静态构造函数也可以有实例构造函数。
- ❑ 如同静态方法，静态构造函数不能访问所在类的实例成员，因此也不能使用this访问器。
- ❑ 不能从程序中显式调用静态构造函数，它们被系统自动调用。
  - ■ 在类的任何实例被创建之前。
  - ■ 在类的任何静态成员被引用之前。

### 6.10.1 静态构造函数示例

下面的代码使用静态构造函数初始化一个名称为RandomKey的Random型私有静态字段。Random

是由BCL提供的用于产生随机数的类。它在System命名空间中。

```
class RandomNumberClass
{
   private static Random RandomKey;          // 私有静态字段

   static RandomNumberClass()                // 静态构造函数
   {
      RandomKey = new Random();              // 初始化RandomKey
   }

   public int GetRandomNumber()
   {
      return RandomKey.Next();
   }
}

class Program
{
   static void Main()
   {
      RandomNumberClass a = new RandomNumberClass();
      RandomNumberClass b = new RandomNumberClass();

      Console.WriteLine("Next Random #: {0}", a.GetRandomNumber());
      Console.WriteLine("Next Random #: {0}", b.GetRandomNumber());
   }
}
```

这段代码的其中一次执行产生以下输出：

```
Next Random #: 47857058
Next Random #: 1124842041
```

### 6.10.2 构造函数的可访问性

可以为实例构造函数指派访问修饰符，就像对其他成员所做的那样。注意，示例中构造函数被声明为public，这样可以从类的外部创建实例。

也可以创建private构造函数，它不能从类的外部调用，但可以从类的内部调用，你将在下一章看到。

## 6.11 对象初始化列表

对象初始化列表允许在创建新的对象实例时设置字段和属性的值。

在此之前的内容中你已经看到，对象创建表达式由关键字new后面跟着一个类构造函数组成。对象初始化列表扩展了创建语法，在表达式的尾部放置了一组成员初始化列表。该语法有两种形式，如下面所示。一种形式包括构造函数的参数列表，另一种不包括。

```
                                 对象初始化列表
                                       ↓
new TypeName(ArgList) { FieldOrProp = InitExpr, FieldOrProp = InitExpr, ...}
new TypeName          { FieldOrProp = InitExpr, FieldOrProp = InitExpr, ...}
                                  ↑                        ↑
                            成员初始化列表            成员初始化列表
```

例如，对于一个名称为Point的类，它有两个公有整型字段X和Y，你可以使用下面的表达式创建一个新对象：

```
new Point { X = 5, Y = 6 };
              ↑      ↑
          初始化X  初始化Y
```

关于对象初始化列表的重要内容如下：

- 被初始化的成员在创建对象的代码中必须是可访问的（如public）。
- 初始化在构造函数完成之后发生。

下面的代码展示了一个使用对象初始化列表的示例。在Main中，pt1只调用构造函数，构造函数设置了它的两个字段的值。然而，对于pt2，构造函数把字段的值设置为1和2，初始化语句把它们改为5和6。

```
public class Point
{
   public int X = 1;
   public int Y = 2;
}

static void Main( )
{                                 对象初始化列表
   Point pt1 = new Point();             ↓
   Point pt2 = new Point      { X = 5, Y = 6 };
   Console.WriteLine("pt1: {0}, {1}", pt1.X, pt1.Y);
   Console.WriteLine("pt2: {0}, {1}", pt2.X, pt2.Y);s
}
```

这段代码产生以下输出：

```
pt1: 1, 2
pt2: 5, 6
```

## 6.12　析构函数

析构函数（destructor）执行在类的实例被销毁之前需要的清理或释放非托管资源的行为。关于析构函数的重要内容如下：

- 每个类只能有一个析构函数。
- 析构函数不能带参数。
- 析构函数不能带访问修饰符。

- 析构函数和类有相同的名称，但以一个“~”字符作前缀（“~”的英文发音为 TIL-duh）。
- 析构函数只对类的实例起作用，因此没有静态析构函数。
- 不能在代码中显式地调用析构函数。相反，它在垃圾收集过程中调用，当时垃圾收集器分析代码，并确定在代码中没有任何途径引用该对象。

例如，下面的代码阐明了名称为Class1的类的析构函数的语法：

```
Class1
{
   ~Class1()                    // 析构函数
   {
      CleanupCode
   }
   ...
}
```

使用析构函数的一些重要指导方针如下：

- 如果不需要，就不要执行析构函数，它们会带来性能上的开销。
- 析构函数只应释放对象自己的外部资源，它不应该访问其他的对象，因为无法假定这些对象还没有被收集。

---

**说明** 尽管析构函数应该被叫做“析构函数”还是“终结函数”(finalizer)还有些争议，但2007年底发布的C#语言规范 3.0版称这个方法为析构函数。

---

### 6.12.1 调用析构函数

与C++析构函数不同，C#析构函数并不在实例失效时立即调用。事实上，无法知道析构函数会在什么时候调用。此外，如前所述，也不能显式地调用析构函数。如果你的代码需要析构函数，你就必须为系统提供它，系统会在对象从托管的堆中移走之前的某点调用它。

如果你的代码中包含需要及时清理的非托管资源，别把它留给析构函数处理，因为不能保证析构函数会很快运行。相反，你应该采用标准模式，在类中实现名称为IDisposable的接口。（接口将在第17章阐述。）接口中把资源的清理代码封装在一个void型的无参数方法中，该方法应该叫做Dispose。

当你使用完资源并想释放它们时，你需要调用Dispose。注意，是你调用Dispose，而不是析构函数。系统不会为你自动调用它。

关于Dispose方法的一些指导方针如下：

- 以一种方法实现Dispose中的代码，使该方法被不止一次调用时也是安全的。如果它已经被调用过了，那么接下来的任何调用都不应该产生异常或做任何附加工作（异常将在第11章阐述）。
- 把Dipose方法和析构函数编写成这样，如果因为某种原因，你的代码没有调用Dispose，你的析构函数应该调用它并释放资源。

- 因为是Dispose做清理而不是析构函数，所以它应当调用GC.SuppressFinalize方法，该方法告诉CLR不要调用该方法的析构函数，因为它已经被清理了。

下面的代码描述了一个安全的清理过程。首先，类需要声明一个布尔型disposed字段以弄明白清理是否已经发生。在对象被创建时它被初始化为false。

在Dispose方法内部，做下面的事情。

- 检查标记，看看资源是否已经被释放了。如果没有，做下面的事：
    - 对每个需要清理的托管资源调用Dispose方法。
    - 释放对象持有的任何非托管资源。
- 现在清理已经发生了，设置disposed标记为true。
- 最后，调用垃圾收集器的SuppressFinalize方法，告诉垃圾收集器不要调用类的析构函数。

在析构函数内的处理过程和Dispose方法内相似，但更短。只是检查对象是否已经被清理，如果没有，就释放非托管资源。注意，在这种情况下，不要对任何托管资源调用Dispose方法，因为垃圾收集器可能已经把这些对象删除了。

```
class MyClass
{
   bool disposed = false;              // 标记指示dispoal状态

   ///////////////////////////////////////////////////////////
   public void Dispose()               // 公有的Dispose
   {
      if (disposed == false)           // 检查标记
      {
         // 在托管资源中调用Dispose
         ...

         // 释放所有未托管资源
         ...
      }
      disposed = true;                 // 设置标记以显示disposal
      GC.SuppressFinalize(this);       // 告知GC不要调用终结函数
   }

   ///////////////////////////////////////////////////////////
   ~MyClass()                          // Destructor
   {
      if (disposed == false)           // 检查标记
      {
         // Release any unmanaged resources.
         ...
      }
   }

   ...
}
```

## 6.12.2 标准清理模式

在上一节中，你看到析构函数代码本质上是Dispose代码的子集。标准模式把两个方法的大部分公用代码分解到另外一个名称为Dispose的方法中，我将称这个方法为分解的Dispose。它带一个单独的布尔型参数，用于标识方法是从公有的Dispose方法（true）中调用还是从析构函数（false）调用。

这个标准清理模式如下所示，并在图6-13中阐明。我将在下一章阐述protected和virtual修饰符。

```
class MyClass
{
   bool disposed = false;                              // Disposal状态

   public void Dispose()                               ┐
   {                                                   │
      Dispose( true );                                 │ 公有的Dispose
      GC.SuppressFinalize(this);                       │
   }                                                   ┘

   ~MyClass()                                          ┐
   {                                                   │ 析构函数
      Dispose(false);                                  │
   }                                                   ┘

   protected virtual void Dispose(bool disposing)      ┐
   {                                                   │
      if (disposed == false)                           │
      {                                                │
         if (disposing == true)                        │
         {                                             │
            // Dispose the managed resources.          │ 分解的Dispose
            ...                                        │
         }                                             │
                                                       │
         // Dispose the unmanaged resources.           │
         ...                                           │
      }                                                │
      disposed = true;                                 │
   }                                                   ┘

}
```

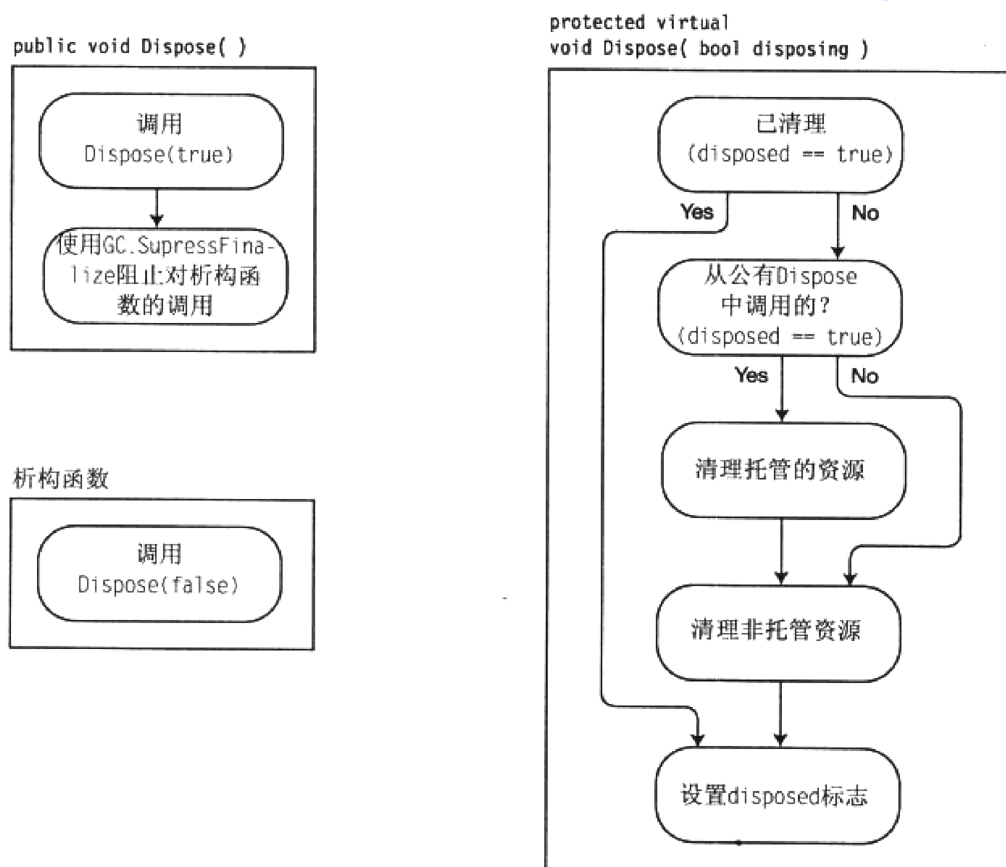



图6-13　标准清理模式

## 6.13　比较构造函数和析构函数

表6-3总结了构造函数和析构函数的调用时间。

表6-3　构造函数和析构函数

| | | 调用时间和调用频度 |
|---|---|---|
| 实例 | 构造函数 | 在创建类的每个新实例时调用一次 |
| | 析构函数 | 对类的每个实例调用，在程序不再访问该实例之后的某一时间点 |
| 静态 | 构造函数 | 只调用一次，在类的任意静态变量第一次被访问之前，或在类的任何实例被创建之前，无论两者谁先发生 |
| | 析构函数 | 不存在，只有实例才有析构函数 |

## 6.14　readonly修饰符

字段可以用readonly修饰符声明。其作用类似于声明一个字段为const，一但值被设定就不能改变。

- ❑ const字段只能在字段的声明语句中初始化，而readonly字段可以在下列任意位置设置它的值：
  - ■ 字段声明语句，如同const。
  - ■ 类的任何构造函数。如果是static字段，初始化必须在static构造函数中完成。
- ❑ const字段的值必须在编译期决定，而readonly字段的值可以在运行期决定。这种增加的自由性允许你在不同的构造函数中设置不同的值！
- ❑ 和const不同，const总是像静态的，而对于readonly字段：
  - ■ 它可以是实例字段，也可以是静态字段。
  - ■ 它在内存中有存储位置。

例如，下面的代码声明了一个名称为Shape的类，它有两个readonly字段。

- ■ 字段PI在它的声明中初始化。
- ■ 字段NumberOfSides被设置为3或4，取决于哪个构造函数被调用。

```
class Shape
{   关键字              初始化
       ↓                   ↓
   readonly double PI = 3.1416;
   readonly int    NumberOfSides;
       ↑                   ↑
    关键字              未初始化

   public Shape(double side1, double side2)                    // 构造函数
   {
      // Shape表示一个矩形
      NumberOfSides = 4;
               ↑
      ……在构造中设定
   }

   public Shape(double side1, double side2, double side3)      // 构造函数
   {
      // Shape表示一个三角形
      NumberOfSides = 3;
               ↑
      ……在构造中设定
   }
}
```

## 6.15 this 关键字

this关键字在类中使用，是对当前实例的引用。它只能被用在下列类成员的代码块中：

- ❑ 实例构造函数。

- 实例方法。
- 属性和索引的实例访问器（索引将在下一节阐述）。

很明显，因为静态成员不是实例的一部分，所以不能在任何静态函数成员的代码中使用this关键字。更适当地说，this被用于下列目的：

- 用于区分类的成员和本地变量，或参数。
- 作为调用方法的实参。

例如，下面的代码声明了类MyClass，它有一个int字段和一个方法，方法带有一个单独的int参数。方法比较参数和字段的值并返回其中较大的值。唯一的问题是字段和形参的名称相同，都是Var1。在方法内使用this关键字引用字段，以区分这两个名称。注意，不推荐参数和类型的字段使用相同的名称。

```
class MyClass
{
   int Var1 = 10;
       ↑两者名称都是“Var1”   ↓
   public int ReturnMaxSum(int Var1)
   {            参数      字段
                 ↓         ↓
      return Var1 > this.Var1
                  ? Var1              // 参数
                  : this.Var1;        // 字段
   }
}

class Program
{
   static void Main()
   {
      MyClass mc = new MyClass();
      Console.WriteLine("Max: {0}", mc.ReturnMaxSum(30));
      Console.WriteLine("Max: {0}", mc.ReturnMaxSum(5));
   }
}
```

## 6.16　索引

如果定义类Employee，它带有三个string型字段（如图6-14所示），那么可以使用字段的名称访问它们，如Main中的代码所示。

然而有的时候，如果能使用索引访问它们将会很方便，好像该实例是字段的数组一样。这正是索引允许做的事。如果为类Employee写一个索引，方法Main看起来就像图6-15中的代码那样。请注意没用使用点运算符，相反，索引使用**索引运算符**，它由一对方括号和中间的索引值组成。

```
class Employee
{
   public string LastName;
   public string FirstName;
   public string CityOfBirth;
}

class Program
{
   static void Main()
   {
      Employee emp1 = new Employee();

      emp1.LastName = "Doe";
      emp1.FirstName = "Jane";
      emp1.CityOfBirth = "Dallas";
      Console.WriteLine("{0}", emp1.LastName);
      Console.WriteLine("{0}", emp1.FirstName);
      Console.WriteLine("{0}", emp1.CityOfBirth);
   }
}
```

字段名

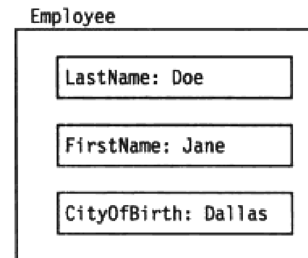



图6-14 没有索引的简单类

```
static void Main()
{
   Employee emp1 = new Employee();

   emp1[0] = "Doe";
   emp1[1] = "Jane";
   emp1[2] = "Dallas";
   Console.WriteLine("{0}", emp1[0]);
   Console.WriteLine("{0}", emp1[1]);
   Console.WriteLine("{0}", emp1[2]);
}
```

索引

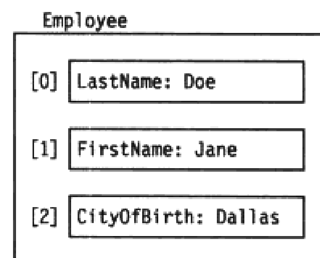



图6-15 使用索引字段

## 6.16.1 什么是索引

索引是一组get和set访问器，类似于属性的访问器。图6-16展示了一个类的索引的形态，该类可以获取和设置string型值。

```
string this [ int index ]
{
   set
   {
      SetAccessorCode
   }
   get
   {
      GetAccessorCode
   }
}
```

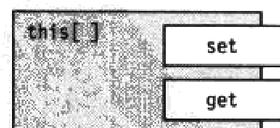



图6-16 索引的形态

### 6.16.2 索引和属性

索引和属性在很多方面是相似的。

- 和属性一样，索引不用分配内存来存储。
- 索引和属性都主要被用来访问其他数据成员，这些成员和它们关联，它们为这些成员提供设置和获取访问。
  - 属性通常访问单独的数据成员。
  - 索引通常访问多个数据成员。

---

**说明** 可以把索引想象成提供获取和设置类的多个数据成员的属性。通过提供索引在许多可能的数据成员中进行选择。索引本身可以是任何类型的，不仅仅是数值类型。

---

使用索引时，另外还有一些注意事项如下：

- 索引可以只有一个访问器，也可以两个都有。
- 索引总是实例成员。因此，索引不能被声明为static。
- 和属性一样，实现get和set访问器的代码不必一定要关联到某个字段或属性。这段代码可以做任何事情或什么也不做，只要get访问器返回某个指定类型的值即可。

### 6.16.3 声明索引

声明索引的语法如下。请注意以下几点：

- 索引没有名称。在名称的位置是关键字this。
- 参数列表在方括号中间。
- 参数列表中至少必须声明一个参数。

```
            关键字      参数列表
              ↓            ↓
ReturnType this [ Type param1, ... ]
{                ↑                  ↑
   get                 方括号
   {
        ...
   }
   set
   {
        ...
   }
}
```

声明索引类似于声明属性。图6-17阐明了它们的句法相似点和不同点。

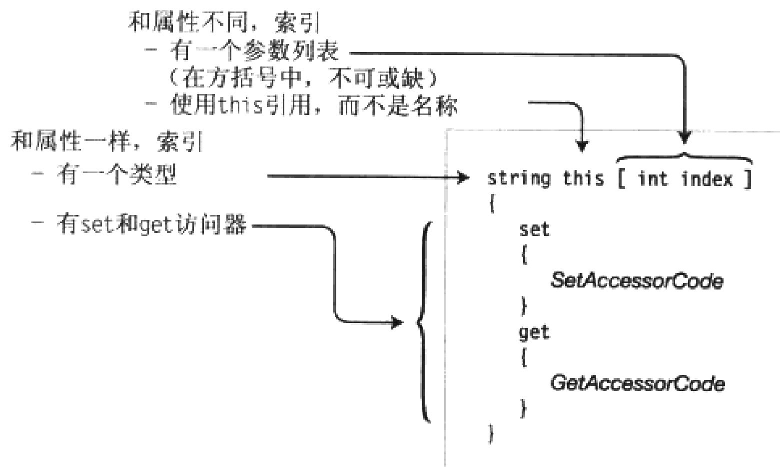



图6-17 比较索引声明和属性声明

## 6.16.4 set 访问器

当索引被用于赋值时，set访问器被调用，并接受两项数据，如下：

- ❑ 一个隐式参数，名称为value，value持有要保存的数据。
- ❑ 一个或更多索引参数，表示数据应该保存到哪里。

```
emp[0] = "Doe";
   ↑       ↑
  索引      值
  参数
```

在set访问器中的代码必须检查索引参数，以确定数据应该存往何处，然后保存它。

set访问器的语法和含义如图6-18所示。图的左边展示了访问器声明的实际语法。右边展示了访问器的语义，把访问器的语义以普通方法的语法书写出来。右边的图例表明set访问器有如下语义：

- ❑ 它的返回类型为void。
- ❑ 它使用的参数列表和索引声明中的相同。
- ❑ 它有一个名称为value的隐式值参，值参类型和索引类型相同。

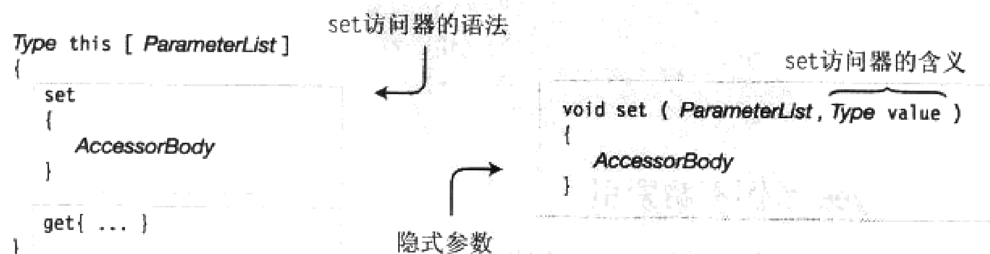



图6-18 set访问器声明的语法和含义

### 6.16.5　get访问器

当索引被用于获取值时，get访问器被一个或多个索引参数调用。索引参数指示哪个值将被获取。

```
string s = emp[0];
                ↑
              索引参数
```

get访问器方法体内的代码必须检查索引参数，确定它表示的是哪个字段，并返回该字段的值。

get访问器的语法和含义如图6-19所示。图的左边展示了访问器声明的实际语法。右边展示了访问器的语义，把访问器的语义以普通方法的语法书写出来。get访问器有如下语义：

- 它的参数列表和索引声明中的相同。
- 它返回与索引相同类型的值。

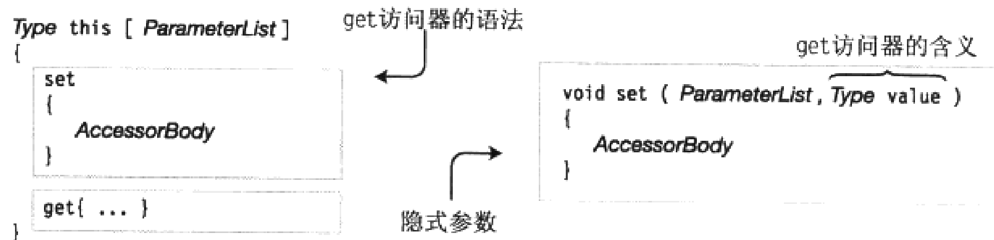



图6-19　get访问器声明的语法和含义

### 6.16.6　关于索引的补充

和属性一样，get和set访问器不能被显示调用。取而代之，当索引被用在表达示中取值时，get访问器自动被调用。当索引被使用赋值语句赋值时，set访问器自动被调用。

当索引被“调用”时，参数在方括号中间提供。

```
    索引    值
     ↓      ↓
emp1[0] = "Doe";                        // 调用set访问器
string NewName = emp[0];                // 调用get访问器
                     ↑
                    索引
```

### 6.16.7　为Employee示例声明索引

下面的代码为先前示例中的类Employee声明一个索引。

- 索引必须读写string类型的值，所以string必须被声明为索引类型。它必须被声明为public以使它能被从类的外部访问。

- 三个字段被强行索引为整数0~2，所以方括号中间的形参必须为int型。本例中形参名称为index。
- 在set访问器方法体内，代码确定索引指的是哪个字段，并把value赋值给它。在get访问器方法体内，代码确定索引指的是哪个字段，并返回该字段的值。

```
class Employee
{
   public string LastName;                          // 调用字段 0
   public string FirstName;                         // 调用字段 1
   public string CityOfBirth;                       // 调用字段 2

   public string this[int index]                    // 索引声明
   {
      set                                           // set访问器声明
      {
         switch (index)
         {
            case 0: LastName = value;
               break;
            case 1: FirstName = value;
               break;
            case 2: CityOfBirth = value;
               break;
            default:
               throw new ArgumentOutOfRangeException("index");
               break;
         }
      }

      get                                           // get访问器声明
      {
         switch (index)
         {
            case 0: return LastName;
            case 1: return FirstName;
            case 2: return CityOfBirth;
            default:
               throw new ArgumentOutOfRangeException("index");
               break;
         }
      }
   }
}
```

### 6.16.8 另一个索引示例

下面是一个附加的示例，索引类Class1的两个int字段。

```
class Class1
{
   int Temp0;                        // 私有字段
   int Temp1;                        // 私有字段
   public int this [ int index ]     // 索引
   {
      get
      {
         return ( 0 == index )       // 返回Temp0或Temp1的值
                    ? Temp0
                    : Temp1;
      }

      set
      {
         if( 0 == index )
            Temp0 = value;           //注意隐式变量"value"
         else
            Temp1 = value;           //注意隐式变量"value"
      }
   }
}

class Example
{
   static void Main()
   {
      Class1 a = new Class1();
      Console.WriteLine("Values -- T0: {0},  T1: {1}", a[0], a[1]);
      a[0] = 15;
      a[1] = 20;
      Console.WriteLine("Values -- T0: {0}, T1: {1}", a[0], a[1]);
   }
}
```

这段代码产生以下输出：

```
Values -- T0: 0,  T1: 0
Values -- T0: 15, T1: 20
```

### 6.16.9　索引重载

只要索引的参数列表不同，类可以有不止一个索引。索引类型不同是不够的。这叫做索引重载，因为所有的索引都有相同的“名称”：this访问引用。

例如，下面的代码有三个索引：两个string类型的和一个int类型的。两个string类型的索引中，一个带单独的int参数，另一个带两个int参数。

```
class MyClass
{
   public string this [ int index ]
   {
      get { ... }
      set { ... }
   }

   public string this [ int index1, int index2 ]
   {
      get { ... }
      set { ... }
   }

   public int this [ float index1 ]
   {
      get { ... }
      set { ... }
   }

   ...
}
```

**说明** 请记住，类中重载的索引必须有不同的参数列表。

## 6.17 访问器的访问修饰符

在这一章中，你已经看到有两种函数成员带get和set访问器：属性和索引。默认情况下，成员的两个访问器有和成员自身相同的访问级别。也就是说，如果一个属性有public访问级别，那么它的两个访问器都有同样的访问级别，对索引也一样。

在特定情况下，成员的访问器可以有不同的访问级别。例如，在下面的代码中，属性Name有public访问级别，但set访问器有protected访问级别。

```
class MyClass
{
   private string _Name = "John Doe";
   public string Name
   {
      get { return _Name; }
      protected set { _Name = value; }
   }
}
```

访问器的访问修饰符有几个限制。最重要的限制如下：

- 仅当成员（属性或索引）既有get访问器也有set访问器时，其访问器才能有访问修饰符。
- 虽然两个访问器都必须出现，但它们中只能有一个有访问修饰符。

- 访问器的访问修饰符必须比成员的访问级别有更严格的限制性。

图6-20阐明了访问级别的层次。访问器的访问级别在图表中的位置必须比成员的访问级别的位置低。

例如，如果一个属性的访问级别是public，在图表里较低的四个级别中，可以把任意一个级别给它的一个访问器。但如果属性的访问级别是protected，唯一能对访问器使用的访问修饰符是private。

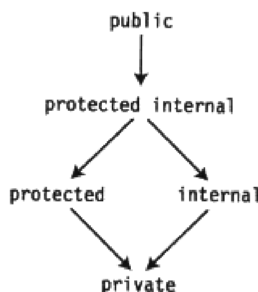



图6-20　访问器级别的限制性层次

## 6.18　分部类和分部类型

类的声明可以被分割成几个分部类的声明。

- 每个分部类的声明都含有一些类成员的声明。
- 类的分部类声明可以在同一文件中也可以在不同文件中。

每个局部声明必须被标为partial class，而不是单独的关键字class。分部类声明看起来和普通类声明相同，除了那个附加的类型修饰符partial。

```
Type modifier

partial class MyPartClass    // 类名称与下面的相同
{
   member1 declaration
   member2 declaration
      ...
}
Type modifier

partial class MyPartClass    // 类名称与上面的相同
{
   member3 declaration
   member4 declaration
      ...
}
```

**说明**　类型修饰符partial不是关键字，所以在其他上下文中，可以在程序中把它用作标识符。但直接用在关键字class、struct或interface之前时，它表示分部类型。

例如，图6-21中左边的框表示一个类声明文件。图右边的框表示相同的类声明被分割成两个文件。

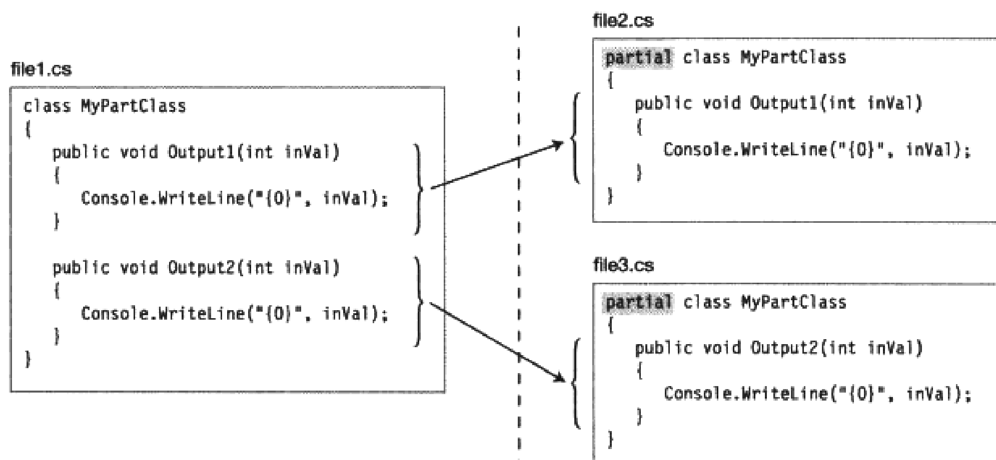



图6-21　使用分部类型分割类

组成类的所有分部类声明必须在一起编译。使用分部类声明的类必须有相同的含义，如同它所有类成员都被声明在一个单独的类声明体内。

除了类，还可以创建两种分部类型：

- 局部结构（结构在第12章阐述）。
- 局部接口（接口在第17章阐述）。

## 分部方法

分部方法是在分部类中声明在两个部分中的方法。分部方法的那两个部分可以声明在分部类的不同部分，也可以声明在同一部分。分部方法的两个部分如下。

- 定义声明给出签名和返回类型，声明的实现部分只是一个分号。
- 实现声明给出签名、返回类型，还有正常形式的语句块实现。

关于分部方法需要了解的重要内容如下。

- 定义声明和实现声明的签名和返回类型必须匹配。签名和返回类型有如下特征：
  - 在定义声明和实现声明中都必须包含上下文关键字partial，直接放在关键字void之前。
  - 签名不能包括访问修饰符，这使分部方法是隐式私有的。
  - 返回类型必须是void。
  - 参数列表不能包含out参数。
- 可以有定义部分而没有实现部分。在这种情况下，编译器把方法的声明以及方法内部任何对方法的调用都移除。然而，如果类有分部方法的实现部分，它也必须有定义部分。

下面的代码展示了一个名称为PrintSum的分部方法的示例。

- PrintSum声明在分部类MyClass的不同部分：定义声明在第一个部分中，实现声明在第二

个部分中。实现部分打印出两个整型参数的和。

- 因为分部方法是隐式私有的，PrintSum不能从类的外部调用。方法Add是调用PrintSum的公有方法。
- Main创建一个类MyClass的对象，并调用它的公有方法Add，Add方法调用方法PrintSum，PrintSum打印出输入参数的和。

```
partial class MyClass
{         必须是void
              ↓
   partial void PrintSum(int x, int y);       // 定义分部方法
         ↑                               ↑
  上下文关键字                        没有实现部分
   public void Add(int x, int y)
   {
      PrintSum(x, y);
   }
}

partial class MyClass
{
   partial void PrintSum(int x, int y)       // 实现分部方法
   {
      Console.WriteLine("Sum is {0}", x + y);      ← 实现部分
   }
}

class Program
{
   static void Main( )
   {
      var mc = new MyClass();
      mc.Add(5, 6);
   }
}
```

这段代码产生以下输出：

```
Sum is 11
```

第 7 章

# 类和继承

**本章内容**

- 类继承
- 访问继承的成员
- 隐藏基类成员
- 基类访问
- 使用基类的引用
- 构造函数的执行
- 程序集间的继承
- 成员访问修饰符
- 抽象成员
- 抽象类
- 密封类
- 静态类
- 扩展方法
- 外部方法

## 7.1 类继承

通过继承我们可以定义一个新类，新类纳入一个已经声明的类并进行扩展。

- 可以使用一个已经存在的类作为新类的基础。已存在的类称为基类（base class），新类称为派生类（derived class）。派生类成员的组成如下：
  - 自己声明中的成员。
  - 基类的成员。
- 要声明一个派生类，需要在类名后加入基类规格说明。基类规格说明由冒号和后面跟着用作基类的类的名称组成。派生类被描述为直接继承自列出的基类。
- 派生类被描述为扩展它的基类，因为它包含了基类的成员，加上在它自己的声明中的任何附加功能。
- 派生类不能删除它所继承的任何成员。

例如，下面展示了名为OtherClass的类的声明，它继承名称为SomeClass的类：

```
                        基类描述符
                            ↓
class OtherClass : SomeClass
{                     ↑       ↑
   ...               冒号    基类
}
```

图7-1展示了每个类的实例。在左边，类SomeClass有一个字段和一个方法。在右边，类OtherClass继承SomeClass，并包含了一个附加的字段和一个附加的方法。

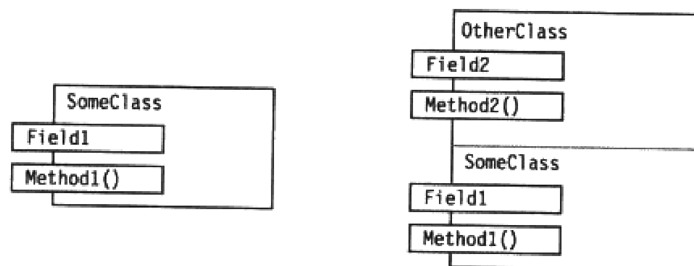



图7-1　基类和派生类

## 7.2　访问继承的成员

继承的成员可以被访问，就像它们是派生类自己声明的一样（继承的结构有些不同，在本章接下来会阐述它们）。例如，下面的代码声明了类SomeClass和OtherClass，它们如图7-1所示。这段代码显示OhterClass的所有四个成员都能被无缝地访问，无论它们是在基类中声明的还是在派生类中声明的。

- Main创建派生类OtherClass的一个对象。
- Main中接下来的两行调用基类中的Method1，先是使用基类的Field1，然后是派生类的Field2。
- Main中后续的两行调用派生类中的Method2，再次先使用基类的Field1，然后是派生类的Field2。

```
class SomeClass {                              // 基类
   public string Field1 = "base class field";
   public void Method1( string value ) {
      Console.WriteLine("Base class -- Method1:     {0}", value);
   }
}

class OtherClass: SomeClass {                  // 派生类
   public string Field2 = "derived class field";
   public void Method2( string value ) {
      Console.WriteLine("Derived class -- Method2:  {0}", value);
   }
```

```
    }

    class Program {
       static void Main() {
          OtherClass oc = new OtherClass();

          oc.Method1( oc.Field1 );       // 以基类字段为参数的基类方法
          oc.Method1( oc.Field2 );       // 以派生字段为参数的基类方法
          oc.Method2( oc.Field1 );       // 以基类字段为参数的派生方法
          oc.Method2( oc.Field2 );       // 以派生字段为参数的派生方法
       }
    }
```

这段代码产生以下输出：

```
Base class -- Method1:     base class field
Base class -- Method1:     derived class field
Derived class -- Method2:  base class field
Derived class -- Method2:  derived class field
```

## 所有类都派生自 object 类

除了特殊的类object，所有的类都是派生类，即使它们没有基类规格说明。类object是唯一的非派生类，因为它是继承层次结构的基础。

没有基类规格说明的类隐式地直接派生自类object。不加基类规格说明只是指定object为基类的简写。这两种形式是语义等价的。

图7-2展示了同一个类的这两种声明形式。

```
class SomeClass              class SomeClass : object
{                            {
   ...                          ...
}                            }
```

隐式派生自object　　　显式派生自object

图7-2　直接继承object

关于类继承的其他重要内容如下：

- 一个类声明的基类规格说明中只能有一个单独的类。这称为单继承。
- 虽然类只能直接继承一个基类，但继承的层次没有限制。也就是说，作为基类的类可以派生自另外一个类，而它又派生自另外一个类，一直下去，直至最终到达object。

基类和派生类是相对的术语。所有的类都是派生类，要么派生自object，要么派生自其他的类。所以，通常当我们称一个类为派生类时，我们的意思是它直接派生自某类而不是object。图7-3展示了一个简单的类层次结构。在这之后，将不会在图中显示object了，因为所有的类最终都派生自它。

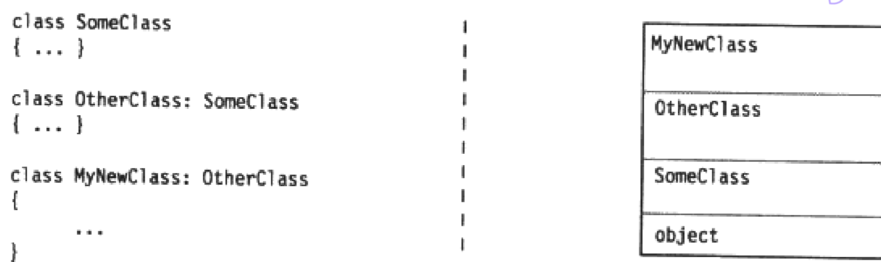



图7-3 类层次结构

## 7.3 隐藏基类的成员

虽然派生类不能删除它继承的任何成员，但它可以隐藏它们。

- 要隐藏一个继承的数据成员，需要声明一个新的相同类型的成员，并使用相同的名称。
- 通过在派生类中声明新的带有相同签名的函数成员，可以隐藏或掩盖继承的函数成员。请记住，签名由名称和参数列表组成，但不包括返回类型。
- 要让编译器知道你在故意隐藏继承的成员，使用new修饰符。没有它，程序可以成功编译，但编译器会警告你隐藏了一个继承的成员。
- 也可以隐藏静态成员。

下面的代码声明了一个基类和一个派生类，它们都有一个名称为Field1的string成员。使用new关键字以显式地告诉编译器掩盖基类成员。图7-4阐明了每个类的实例。

```
class SomeClass                          // 基类
{
   string Field1;
   ...
}

class OtherClass : SomeClass             // 派生类
{
   new string Field1;                    // 用同样的名称掩盖基类成员
    ↑
  关键字
```

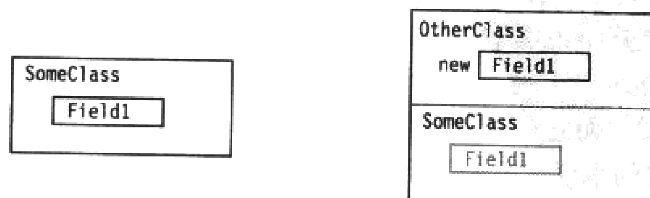



图7-4 隐藏基类成员

在下面的代码中，OtherClass派生自SomeClass，但隐藏了两个继承的成员。注意new修饰符的使用，图7-5阐明了这段代码。

```
class SomeClass                                    // 基类
{
   public string Field1 = "SomeClass Field1";
   public void   Method1(string value)
      { Console.WriteLine("SomeClass.Method1:  {0}", value); }
}

class OtherClass : SomeClass                       // 派生类
{ 关键字
   ↓
   new public string Field1 = "OtherClass Field1";  // 掩盖基类成员
   new public void   Method1(string value)          // 掩盖基类成员
   ↑   { Console.WriteLine("OtherClass.Method1:  {0}", value); }
}
  关键字

class Program
{
   static void Main()
   {
      OtherClass oc = new OtherClass();        // 使用掩盖成员
      oc.Method1(oc.Field1);                   // 使用掩盖成员
   }
}
```

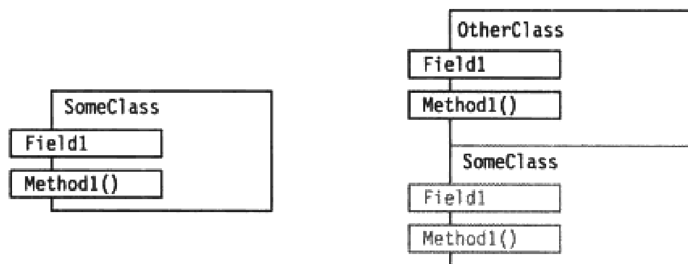



图7-5　隐藏基类的字段和方法

## 7.4　基类访问

有时，派生类需要访问被隐藏的继承成员。可以使用**基类访问**表达式访问隐藏的基类成员。基类访问表达式由关键字base后面跟着一个点和成员的名称组成，如下所示：

```
Console.WriteLine("{0}", base.Field1);
                         ↑
                      基类访问
```

例如，在下面的代码中，派生类OtherClass隐藏了基类的Field1，但是使用基类访问表达式访问它。

```
class SomeClass {                                          // 基类
   public string Field1 = "Field1 -- In the base class";
}

class OtherClass : SomeClass {                             // 派生类

   new public string Field1 = "Field1 -- In the derived class";
   ↑                ↑
     隐藏了基类中的字段
   public void PrintField1()
   {
      Console.WriteLine(Field1);                // 访问派生类
      Console.WriteLine(base.Field1);           // 访问基类
   }                       ↑
}                       基类访问

class Program {
   static void Main()
   {
      OtherClass oc = new OtherClass();
      oc.PrintField1();
   }
}
```

这段代码产生以下输出：

```
Field1 -- In the derived class
Field1 -- In the base class
```

## 7.5　使用基类的引用

派生类的实例由基类的实例加上派生类附加的成员组成。派生类的引用指向整个类对象，包括基类部分。

如果有一个派生类对象的引用，就可以获取该对象基类部分的引用，使用类型转换运算符把该引用转换为基类类型。类型转换运算符放置在对象引用的前面，由圆括号括起的要被转换成的类名组成。类型转换将在第18章阐述。

接下来的几节将阐述使用对象的基类部分的引用访问对象。我们从观察下面两行代码开始，它们声明了对象的引用。图7-6阐明了代码，并展示了不同变量所看到的对象部分。

- 第一行声明并初始化了变量derived，它包含一个MyDerivedClass类型对象的引用。
- 第二行声明了一个基类类型MyBaseClass的变量，并把derived中的引用转换为该类型，给出对象的基类部分的引用。
  - 基类部分的引用被储存在变量mybc中，在赋值运算符的左边。
  - 其他部分的引用不能“看到”派生类对象的其余部分，因为它通过基类类型的引用“看”这个对象。

```
MyDerivedClass derived = new MyDerivedClass();      // 创建一个对象
MyBaseClass mybc       = (MyBaseClass) derived;     // 转换引用
```

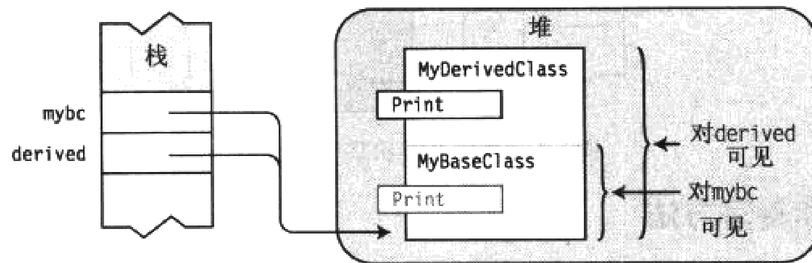



图7-6 派生类的引用可以看到完整的MyDerivedClass对象，而mybc只能看到对象的MyBaseClass部分

下面的代码展示了两个类的声明和使用。图7-7阐明了内存中的对象和引用。

Main创建了一个MyDerivedClass类型的对象，并把它的引用储存到变量derived中。Main还创建了一个MyBaseClass类型的变量，并用它储存对象基类部分的引用。当对每个引用调用Print方法时，调用的是该引用所能看到的方法的实现，并产生不同的输出字符串。

```
class MyBaseClass {
   public void Print() {
      Console.WriteLine("This is the base class.");
   }
}

class MyDerivedClass : MyBaseClass {
   new public void Print() {
      Console.WriteLine("This is the derived class.");
   }
}

class Program {
   static void Main() {
      MyDerivedClass derived = new MyDerivedClass();
      MyBaseClass mybc = (MyBaseClass)derived;
                              ↑
                           转换成基类

      derived.Print();              // 从派生类部分调用Print
      mybc.Print();                 // 从基类部分调用Print
   }
}
```

这段代码产生以下输出：

```
This is the derived class.
This is the base class.
```

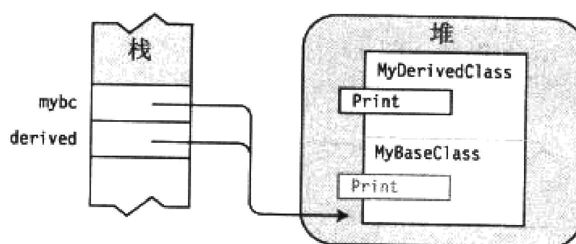



图7-7　对派生类和基类的引用

### 7.5.1　虚方法和覆写方法

在上一节看到，当使用基类引用访问派生类对象时，得到的是基类的成员。虚方法可以使基类的引用访问“升至”派生类内。

可以使用基类引用调用派生类（derived class）的方法，只需满足下面的条件：

- 派生类的方法和基类的方法有相同的签名和返回类型。
- 基类的方法使用virtual标注。
- 派生类的方法使用override标注。

例如，下面的代码展示了基类方法和派生类方法的virtual及override修饰符。

```
class MyBaseClass                                  // 基类
{
   virtual public void Print()
      ↑
   ...
class MyDerivedClass : MyBaseClass                 // 派生类
{
   override public void Print()
      ↑
```

图7-8阐明了这组virtual和override方法。注意和上一种情况相比在行为上的区别，在上一种情况中使用new隐藏基类成员。

- 当使用基类引用（mybc）调用Print方法时，方法调用被传递到派生类并执行，因为
  - 基类的方法被标记为virtual。
  - 在派生类中有匹配的override方法。
- 图7-8阐明了这一点，显示了一个从virtual Print方法后面开始，并指向override Print方法的箭头。

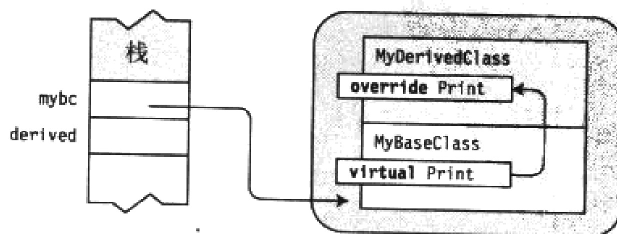



图7-8　虚方法和覆写方法

下面的代码和上一节中的相同，但这一次，方法上标注了virtual和override。产生的结果和前一个示例非常不同。在这个版本中，对基类方法的调用调用了子类中的方法。

```
class MyBaseClass {
   virtual public void Print()
   {
      Console.WriteLine("This is the base class.");
   }
}

class MyDerivedClass : MyBaseClass {
   override public void Print()
   {
      Console.WriteLine("This is the derived class.");
   }
}

class Program {
   static void Main()
   {
      MyDerivedClass derived = new MyDerivedClass();
      MyBaseClass mybc = (MyBaseClass)derived;
                               ↑
      derived.Print();    强制转换成基类
      mybc.Print();
   }
}
```

这段代码产生以下输出：

```
This is the derived class.
This is the derived class.
```

其他关于virtual和override修饰符的重要信息如下：

- 覆写和被覆写的方法必须有相同的可访问性。换一种说法，被覆写的方法不能是private等，而覆写方法是public。
- 不能覆写static方法或非虚方法。
- 方法、属性和索引（在前一章阐述），以及称为事件（将在后面阐述）的另一种成员类型，它们都可以被声明为virtual和override。

### 7.5.2 覆写标记为 override 的方法

覆写方法可以在继承的任何层次出现。

- 当使用对象基类部分的引用调用一个覆写的方法时，方法的调用被沿派生层次上溯执行，一直到标记为override的方法的最派生（most-derived）版本。
- 如果在更高的派生级别有该方法的其他声明，但没有被标记为override，那么它们不会被调用。

例如，下面的代码展示了三个类，形成一个继承的层次：MyBaseClass、MyDerivedClass和SecondDerived。所有这三个类都包含名称为Print的方法，并带有相同的签名。在MyBaseClass中，Print被标记为virtual。在MyDerivedClass中，它被标记为override。在类SecondDerived中，使用override或new声明方法Print。让我们看一看在每种情况下将发生什么。

```
class MyBaseClass                                    // 基类
{
   virtual public void Print()
   { Console.WriteLine("This is the base class."); }
}

class MyDerivedClass : MyBaseClass                   // 派生类
{
   override public void Print()
   { Console.WriteLine("This is the derived class."); }
}

class SecondDerived : MyDerivedClass                 // 最派生类
{
   ... // Given in the following pages
}
```

### 1. 情况1：使用override声明Print

如果把SecondDerived的Print方法声明为override，那么它会覆写方法的全部两个低继承级别的版本，如图7-9所示。如果一个基类的引用被用于调用Print，它会向上传递通过整个链达到类SecondDerived中的实现。

下面的代码实现了这种情况。注意方法Main的最后两行中的代码。

- 两条语句中的第一条使用最派生类SecondDerived的引用调用Print方法。这不是通过基类部分的引用的调用，所以它将会调用SecondDerived中实现的方法。
- 然而，第二条语句使用基类MyBaseClass的引用调用Print方法。

```
class SecondDerived : MyDerivedClass {
   override public void Print() {
      ↑   Console.WriteLine("This is the second derived class.");
   }
}

class Program {
   static void Main()
   {
      SecondDerived derived = new SecondDerived(); // 使用SecondDerived
      MyBaseClass mybc = (MyBaseClass)derived;     // 使用MyBaseClass

      derived.Print();
      mybc.Print();
   }
}
```

结果是：无论Print是通过派生类调用还是通过基类调用，都是最派生类中的方法被调用。当通过基类调用时，调用被沿着继承层次向上传递。这段代码产生以下输出：

```
This is the second derived class.
This is the second derived class.
```

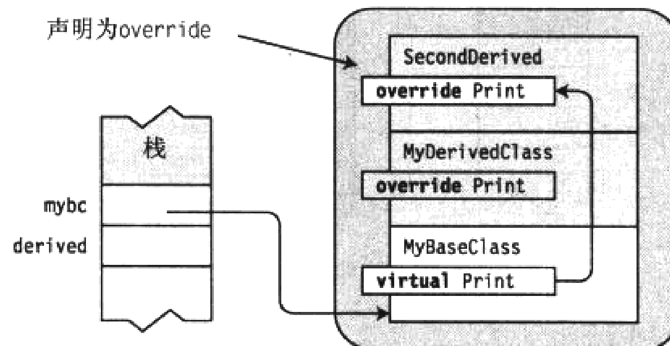



图7-9 执行被传递到多层覆写链的顶端

### 2. 情况2：使用new声明Print

相反，如果声明SecondDerived中的Print方法为new，则结果如图7-10所示。Main和上一种情况中相同。

```
class SecondDerived : MyDerivedClass {
   new public void Print() {
      Console.WriteLine("This is the second derived class.");
   }
}

class Program {
   static void Main()                                     // Main
   {
      SecondDerived derived = new SecondDerived();        // 使用SecondDerived
      MyBaseClass mybc = (MyBaseClass)derived;            // 使用MyBaseClass

      derived.Print();
      mybc.Print();
   }
}
```

结果是：当方法Print通过SecondDerived的引用调用时，SecondDerived中的方法被执行，正如所期待的那样。然而，当方法通过MyBaseClass的引用调用时，方法调用只向上传递了一级，到达类MyDerived，在那里它被执行。两种情况的唯一不同是SecondDerived中的方法使用修饰符override还是修饰符new声明。

这段代码产生以下输出：

```
This is the second derived class.
This is the derived class.
```

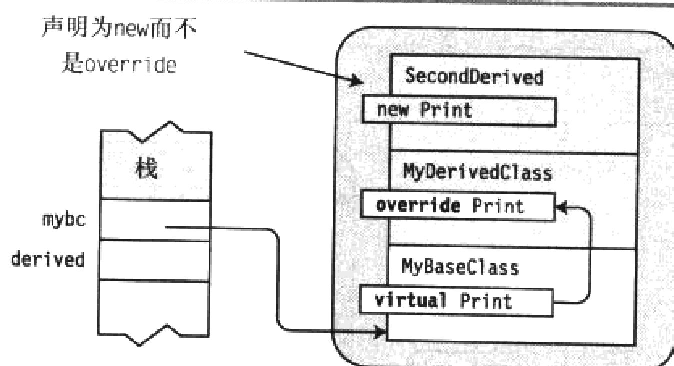



图7-10 隐藏覆写的方法

## 7.6 构造函数的执行

在前面的章节中，你看到构造函数执行代码，准备一个类以使用。这包括初始化类的静态成员和实例成员。在这一章，你会看到派生类对象有一部分就是基类对象。

- 要创建对象的基类部分，基类的一个构造函数被作为创建实例过程的一部分被调用。
- 继承层次链中的每个类在执行它自己的构造函数体之前执行它的基类的构造函数。

例如，下面的代码展示了类MyDerivedClass和它的构造函数的声明。当该构造函数被调用时，它在执行自己的方法体之前调用无参数的构造函数MyBaseClass()。

```
class MyDerivedClass : MyBaseClass
{
   MyDerivedClass()          // 构造函数调用基类构造函数MyBaseClass()
   {
      ...
   }
```

构造的顺序如图7-11所示。当一个实例被创建时，完成的第一件事情是初始化对象的所有实例成员。在此之后，基类的构造函数被调用，然后该类自己的构造函数体才被执行。

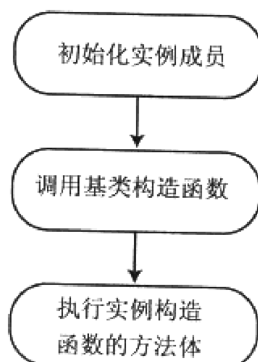



图7-11 对象构造的顺序

例如，在下面的代码中，MyField1和MyField2的值在基类构造函数被调用之前会分别被设置为5和0。

```
class MyDerivedClass : MyBaseClass
{
   int MyField1 = 5;                    // 1. 成员初始化
   int MyField2;                        //    成员初始化

   public MyDerivedClass()              // 3. 构造函数体执行
   {
      ...
   }
}

class MyBaseClass
{
   public MyBaseClass()                 // 2. 基类构造函数调用
   {
      ...
   }
}
```

**警告** 在构造函数中调用虚方法是极不推荐的。在基类的构造函数被执行时，基类的虚方法会调用派生类的覆写方法，但这是在派生类的构造函数方法体被执行之前。因此，调用会在派生类没有完全初始化之前传递到派生类。

### 7.6.1 构造函数初始化语句

默认情况下，在对象被构造时，基类的无参数构造函数被调用。但构造函数可以被重载，所以基类可能有一个以上的构造函数。如果希望派生类使用一个指定的基类构造函数而不是无参数构造函数，必须在构造函数初始化语句中指定它。

有两种形式的构造函数初始化语句：

- 第一种形式使用关键字base并指明使用哪一个基类构造函数。
- 第二种形式使用关键字this并指明应该使用当前类的哪一个另外的构造函数。

基类构造初始化语句放在冒号后面，冒号紧跟着类的构造函数声明的参数列表。构造函数初始化语句由关键字base和要调用的基类构造函数的参数列表组成。

例如，下面的代码展示了类MyDerivedClass的构造函数。

- 构造函数初始化语句指明要使用的基类构造函数是有两个参数的那个。第一个参数是一个string，第二个参数是一个int。
- 在基类参数列表中的参数必须在类型和顺序方面匹配已定的基类构造函数的参数列表。

```
                                        构造函数初始化语句
                                               ↓
public MyDerivedClass( int x, string s ) : base( s, x )
{                                        ↑
   ...                                 关键字
```

当声明一个不带构造函数初始化语句的构造函数时，它是带有base()组成的构造函数初始化语句的形式的简写，如图7-12所阐明的。这两种形式是语义等价的。

```
class MyDerived: MyBase              |  class MyDerived: MyBase
{                                    |  {
   MyDerived()                       |     MyDerived() : base()
   {                                 |     {
      ...                            |        ...
   }                                 |     }
   ...                               |     ...
隐式使用基类构造函数                  |  显式使用基类构造函数
MyBase()的构造函数。                  |  MyBase()的构造函数。
```

图7-12 等价的构造函数形式

另一种构造函数初始化语句的形式指示构造函数使用同一类的不同构造函数。例如，下面的代码展示了类MyClass的构造函数，它使用了同一类中但有两个参数的构造函数，对第二个参数提供了默认参数。

```
                          构造函数初始化语句
                                  ↓
public MyClass(int x): this(x, "Using Default String")
{                         ↑
   ...                  关键字
}
```

## 7.6.2 类访问修饰符

类可以被系统中其他类看到并访问。这一节阐述类的可访问性。虽然我会在解说和示例中使用类，因为类是我们在书中一直阐述的内容，但可访问性规则也适用于以后将会阐述到的其他类型。

术语可见的有时用作述语可访问的。它们可以被互换使用。类的可访问性有两个级别：public和internal。

- 标记为public的类可以被系统内任何程序集中的代码访问。要使一个类对其他程序集可见，使用public访问修饰符，如这里所示：

```
关键字
↓
public class MyBaseClass
{ ...
```

- 标记为internal的类只能被它自己所在的程序集内的类看到。

- 这是默认的可访问级别，所以，除非在类的声明中显式地指定修饰符public，程序集外部的代码不能访问该类。
- 可以使用internal访问修饰符显式地声明一个类为内部的。

```
关键字
  ↓
internal class MyBaseClass
{ ...
```

图7-13阐明了internal和public类从程序集的外部的可访问性。类MyClass对左边程序集内的类不可见，因为它被标记为internal。然而，类OtherClass对于左边的类可见，因为它被标记为public。

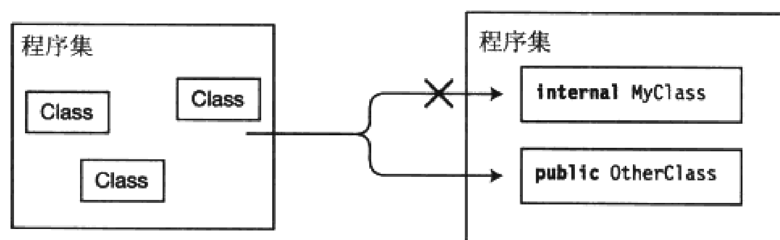



图7-13 其他程序集中的类可以访问公有类但不能访问内部类

## 7.7 程序集间的继承

迄今为止，一直在基类声明的同一程序集内声明派生类。但C#也允许从一个在不同的程序集内定义的基类派生类。要做到这一点，必须满足下列条件：

- 基类必须被声明为public，这样它才能被从它所在的程序集外部访问。
- 必须在Visual Studio工程中包括对包含该基类的程序集的引用。

为了使引用其他程序集中的类和类型更容易，不使用它们的完全限定名称，在源文件的顶部放置一个using指令，并带上将要访问的类或类型所在的命名空间。

**说明** 增加对其他程序集的引用和增加using指令是两回事。增加对其他程序集的引用是告诉编译器所需的类型在哪里被定义。增加using指令允许你引用其他的类而不必使用它们的完全限定名称。第10章会详细阐述这部分内容。

例如，下面两个在不同的程序集中的代码片段展示了继承一个其他程序集中的类是多么容易。第一段代码创建了含有名称为MyBaseClass类的程序集，该类有以下特征：

- 它声明在名称为Assembly1.cs的源文件中，并在声明为BaseClassNS的命名空间的内部。
- 它被声明为public，这样它就可以被从其他程序集中访问。

- ❑ 它含有一个单独的成员，一个名称为PrintMe的方法，仅写出一条简单的消息标识该类。

```
// 源文件名称为Assembly1.cs
using System;
        包含基类声明的命名空间
                ↓
namespace BaseClassNS
{
    把该类声明为公有的，以使它能被从程序集的外部看见
    ↓
   public class MyBaseClass {
      public void PrintMe() {
         Console.WriteLine("I am MyBaseClass");
      }
   }
}
```

第二个程序集包含名称为DerivedClass的类的声明，它继承在第一个程序集中声明的MyBaseClass。该源文件名称为Assembly2.cs。图7-14阐明了这两个程序集。

- ❑ DerivedClass的类体为空，但从MyBaseClassr继承了方法PrintMe。
- ❑ Main创建了一个类型DerivedClass的对象并调用它继承的方法PrintMe。

```
// Source file name Assembly2.cs
using System;
using BaseClassNS;
          ↑
    包含基类声明的命名空间
namespace UsesBaseClass
{                          在其他程序集中的基类
                                ↓
   class DerivedClass: MyBaseClass {
      // Empty body
   }

   class Program {
      static void Main( )
      {
         DerivedClass mdc = new DerivedClass();
         mdc.PrintMe();
      }
   }
}
```

这段代码产生以下输出：

```
I am MyBaseClass
```

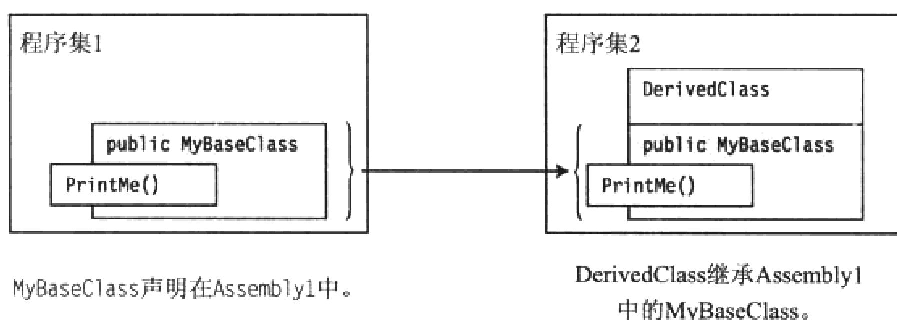



图7-14 跨程序集继承

## 7.8 成员访问修饰符

本章早些时候阐述了类的可访问性。对类的可访问性，只有两种修饰符：internal和public。这一节阐述成员的可访问性。类的可访问性描述了类的可见性；成员的可访问性描述了类成员的可见性。

声明在类中的每个成员对系统的不同部分可见，这依赖于类声明中指派给它的访问修饰符。你已经看到private成员仅对同一类的其他成员可见，而public成员对程序集外部的类也可见。在这一节，我们将再次观察public和private访问级别，以及其他三个可访问性级别。

在观察成员访问性的细节之前，首先有一些通用内容需要阐述：

- 所有显式声明在类的声明中的成员都是互相可见的，无论它们的访问性说明如何。
- 继承的成员不在类的声明中显式声明，所以，如你应该看出，继承的成员对派生类的成员可以是可见的，也可以是不可见的。
- 有五个成员访问级别：
  - 公有的
  - 私有的
  - 受保护的
  - 内部的
  - 受保护内部的
- 必须对每个成员指定成员访问级别。如果不指定某个成员的访问级别，它的隐式访问级别为private。
- 成员不能比它的类更可访问。也就是说，如果一个类的可访问性限于它所在的程序集，那么类的成员个体也不能从程序集的外部看到，无论它们的访问修饰符是什么。

### 7.8.1 访问成员的区域

类声明中的成员访问修饰符详细说明了哪些其他的类型能以及不能访问类的哪些成员。例如，下面的声明展示了使用五种访问级别的成员声明。

```
public class MyClass
{
   public             int Member1;
   private            int Member2;
   protected          int Member3;
   internal           int Member4;
   protected internal int Member5;
   ...
```

访问级别基于两个关于正在声明的类的特征：

- 类是否派生自正在声明的类。
- 类是否和正在声明的类在同一程序集。

这两个特征划分出四个集合，如图7-15所示。与正在声明的类相比，另外的类可以是下面任意一种：

- 在同一程序集并继承它（右下）。
- 在同一程序集但不继承它（左下）。
- 在不同的程序集并继承它（右上）。
- 在不同的程序集但不继承它（左上）。

这些特征被用于定义五种访问级别。

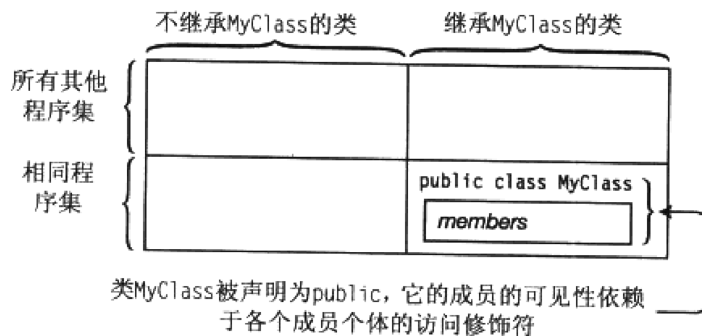



图7-15 访问性的区域划分

## 7.8.2 公有成员的可访问性

public访问级别是限制性最少的。所有的类，包括程序集内部的类和外部的类都可以自由地访问成员。图7-16阐明了MyClass的public类成员的可访问性。

要声明一个公有成员，使用public访问修饰符，如：




```
public int Member1;
```

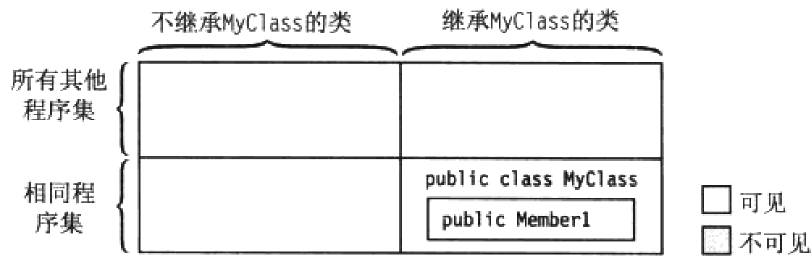


图7-16 公有类的公有成员对同一程序集或其他程序集的所有类可见

### 7.8.3 私有成员的可访问性

私有访问成员是限制最严格的。

- private类成员只能被它自己的类的成员访问。它不能被其他的类访问，包括继承它的类。
- 然而，private成员能被嵌套在它的类中的类的成员访问。嵌套类将在第25章阐述。

图7-17阐明了私有成员的可访问性。

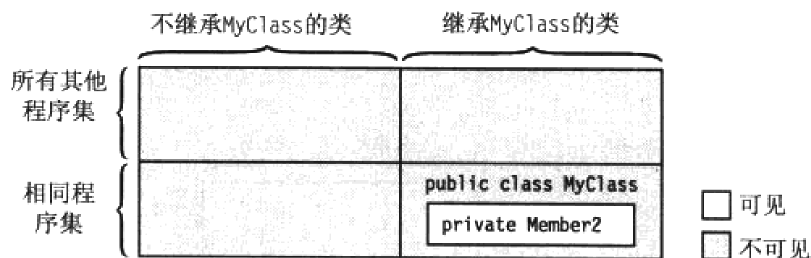


图7-17 任何类的私有成员只对它自己的类（或嵌套类）的成员可见

### 7.8.4 受保护成员的可访问性

protected访问级别如同private访问级别，除了一点，它允许派生自该类的类访问该成员。图7-18阐明了受保护成员的可访问性。注意，即使程序集外部的继承该类的类也能访问该成员。

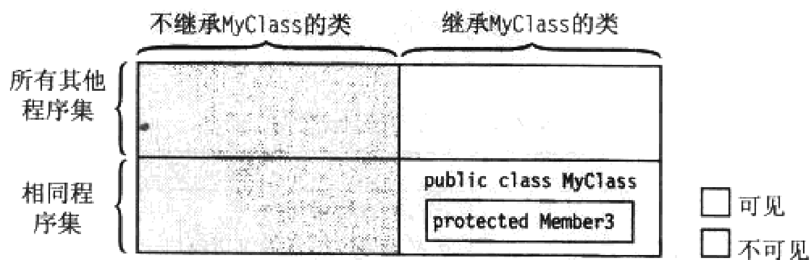


图7-18 公有类的受保护成员对它自己的类的成员或派生类的成员是可见的。派生类甚至可以在其他程序集中

### 7.8.5 内部成员的可访问性

标记为internal的成员对程序集内部的所有类可见，但对程序集外部的类不可见，如图7-19所示。

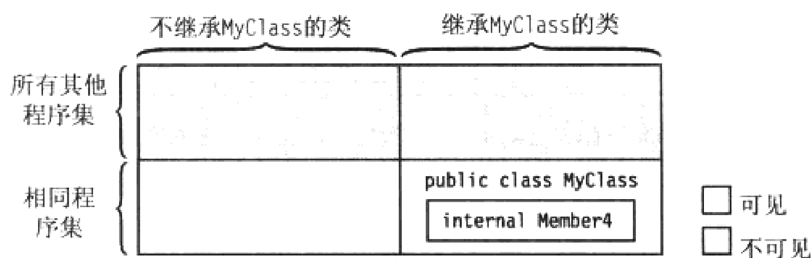



图7-19　内部成员对同一程序集内部的任何类的成员可见，但对程序集外部的类不可见

### 7.8.6 受保护内部成员的可访问性

标记为protected internal的成员对所有继承该类的类以及所有程序集内部的类可见，如图7-20所示。注意，允许访问的集合是protected修饰符允许的类的集合加上internal修饰符允许的类的集合的合集。注意，这是protected和internal的联合，不是交集。

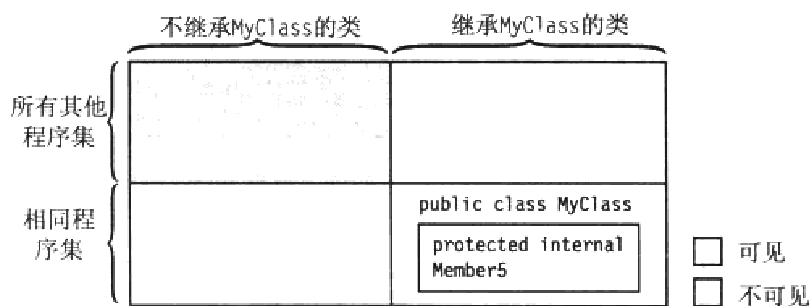



图7-20　公有类的受保护内部成员对相同程序集的类的程序或继承该类的类的成员可见。它对其他程序集中不继承该类的类不可见

### 7.8.7 成员访问修饰符的总结

下面两个表格概括了五种成员访问级别的特征。表7-1列出了修饰符，并直观地概括了修饰符的作用。

**表7-1　成员访问修饰符**

| 修　饰　符 | 含　　义 |
| --- | --- |
| private | 只在类的内部可访问 |
| internal | 对该程序集内所有类可访问 |
| protected | 对所有继承该类的类可访问 |
| protected internal | 对所有继承该类或在该程序集内声明的类可访问 |
| public | 对任何类可访问 |

表7-2在表的左边列出了访问修饰符，并在顶部划分出类的类别。派生指继承声明该成员的类的类。非派生指不继承声明该成员的类的类。表格单元中的对勾的意思是该种类的类可以访问带有相应修饰符的成员。

表7-2 成员可访问性总结

| | 同一程序集内的类 | | 不同程序集内的类 | |
|---|---|---|---|---|
| | 非派生 | 派生 | 非派生 | 派生 |
| private | | | | |
| internal | ✔ | ✔ | | |
| protected | | ✔ | | ✔ |
| protected internal | ✔ | ✔ | | ✔ |
| public | ✔ | ✔ | ✔ | ✔ |

## 7.9 抽象成员

抽象成员是被设计来被覆写的函数成员。抽象成员有以下特征：

- 它被用abstract修饰符标记。
- 它没有实现代码块。抽象成员的代码块被分号代替。

例如，下面取自一个类定义的代码声明了两个抽象成员：一个名称为PrintStuff的抽象方法和一个名称为MyProperty的抽象属性。注意在实现块位置的分号。

```
 关键字                                分号替换实现
   ↓                                       ↓
abstract public void PrintStuff(string s);

abstract public int MyProperty
{
   get;  ← 分号替换实现
   set;  ← 分号替换实现
}
```

关于抽象成员的其他重要事实如下：

- 尽管抽象方法必须在派生类中用相应的方法覆写，但不能把virtual修饰符附加到abstract修饰符。
- 就像虚方法，派生类中抽象方法的实现必须指定override修饰符。
- 抽象成员只能被声明在抽象类中，我们将会在下一节中看到抽象类。

表7-3比较并对照了虚成员和抽象成员。

表7-3 比较虚成员和抽象成员

| | 虚成员 | 抽象成员 |
|---|---|---|
| 关键字 | virtual | abstract |
| 实现体 | 有实现体 | 没有实现体，被分号取代 |

（续）

| | 虚成员 | 抽象成员 |
|---|---|---|
| 在派生类中被覆写 | 能被覆写，使用override | 必须被覆写，使用override |
| 成员的类型 | 方法<br>属性<br>事件<br>索引 | 方法<br>属性<br>事件<br>索引 |

## 7.10　抽象类

抽象类只能被用作其他类的基类。抽象类就是被设计来被继承的。

- 不能创建抽象类的实例。
- 抽象类使用abstract修饰符声明。

```
  关键字
    ↓
abstract class MyClass
{
   ...
}
```

- 抽象类可以包含抽象成员，但不是必须的。抽象类的成员可以是抽象成员和普通带实现的成员的任意组合。
- 抽象类自己可以派生自另一个抽象类。例如，下面的代码展示了一个派生自另一个抽象类的抽象类。

```
abstract class AbClass                    // 抽象类
{
   ...
}

abstract class MyAbClass : AbClass        // 派生自抽象类的抽象类
{
   ...
}
```

- 任何派生自抽象类的类必须使用override关键字实现该类所有的抽象成员，除非派生类自己也是抽象类。

### 抽象类和抽象方法的示例

下面的代码展示了一个名称为AbClass的抽象类，它有两个方法。

第一个方法是一个带有实现的普通方法，它打印出类型的名称。第二个方法是一个必须在派生类中实现的抽象方法。类DerivedClass继承AbClass，而且实现并覆写了抽象方法。Main创建DerivedClass的实例并调用它的两个方法。

```
  关键字
  ↓
abstract class AbClass                                  // 抽象类
{
   public void IdentifyBase()                           // 普通方法
   { Console.WriteLine("I am AbClass"); }
     关键字
     ↓
   abstract public void IdentifyDerived();              // 抽象方法
}

class DerivedClass : AbClass                            // 派生类
{    关键字
     ↓
   override public void IdentifyDerived()               // 抽象方法的实现
   { Console.WriteLine("I am DerivedClass"); }
}

class Example
{
   static void Main()
   {
      // AbClass a = new AbClass();       // 错误，抽象类不能实例化
      // a.IdentifyDerived();

      DerivedClass b = new DerivedClass(); // 实例化派生类
      b.IdentifyBase();                    // 调用继承的方法
      b.IdentifyDerived();                 // 调用“抽象”方法
   }
}
```

这段代码产生以下输出：

```
I am AbClass
I am DerivedClass
```

## 7.11 密封类

在上一节，你看到抽象方法必须被用作基类，它不能像独立的类那样被实例化。密封类与它相反。

- 密封类只能被用作独立的类，它不能被用作基类。
- 密封类使用sealed修饰符标注。

例如，下面的类是一个密封类。任何把它用作其他类的基类的企图都会产生一个编译错误。

```
  关键字
  ↓
sealed class MyClass
{
   ...
}
```

## 7.12 静态类

静态类是这样一种类，在那里所有成员都是静态的。静态类用于分组不受实例数据影响的数据和函数。静态类的一个普通的用途可能就是创建一个包含一组数学方法的数学库。

关于静态类需要了解的重要事情如下：

- 类本身必须标记为static。
- 类的所有成员必须是静态的。
- 类可以有一个静态构造函数，但没有实例构造函数，不能创建该类的实例。
- 不能继承静态类，它们是密封的。

像访问其他静态成员那样访问它的成员，使用类名和成员名。

下面的代码展示了一个静态类的示例：

```
 类必须标记为静态的
   ↓
static public class MyMath
{
   public static float PI = 3.14f;
   public static bool IsOdd(int x)
                ↑       { return x % 2 == 1; }
      成员必须是静态的
                ↓
   public static int Times2(int x)
                       { return 2 * x; }
}

class Program
{
   static void Main( )
   {                                                 使用类名和成员名
      int val = 3;                                          ↓
      Console.WriteLine("{0} is odd is {1}.", val, MyMath.IsOdd(val));
      Console.WriteLine("{0} * 2 = {1}.",     val, MyMath.Times2(val));
   }
}
```

这段代码产生以下输出：

```
3 is odd is True.
3 * 2 = 6.
```

## 7.13 扩展方法

在迄今为止的内容中，你看到的每个方法都和声明它的类关联。C# 3.0的扩展方法特征扩展了这个边界，允许编写和声明它的类之外的类关联的方法。

想要知道可以如何使用这个特征，请看一眼下面的代码。它包含类MyData，该类存储三个

double类型的值，并含有一个构造函数和一个名称为Sum的方法，该方法返回三个存储值的和。

```
class MyData
{
   private double D1;                                       // 字段
   private double D2;
   private double D3;

   public MyData(double d1, double d2, double d3)           // 构造函数
      { D1 = d1; D2 = d2; D3 = d3; }

   public double Sum()                                      // 方法Sum
   {
      return D1 + D2 + D3;
   }
}
```

这是一个非常有限的类，但假设它含有另外一个方法它会更有用，该方法返回三个数据的平均值。使用已经了解的关于类的内容，有几种方法可以实现这个增加的功能：

- 如果你有源代码并可以修改这个类，当然，你仅仅可以为这个类增加一个新方法。
- 然而，如果你不能修改这个类，例如，这个类在一个第三方类库中，那么只要它不是密封的，你就能把它用作一个基类并在派生自它的类中实现这个增加的方法。

然而，如果你不能访问代码，或该类是密封的，或有其他的设计原因使这些方法不能工作，那么你不得不在另一个类中使用该类的公有可用成员编写一个方法。

例如，你可以编写一个像下面代码中这样的类。下面的代码包含一个名称为ExtendMyData的静态类，它含有一个名称为Average的静态方法，该方法实现了增加的功能。注意该方法接受MyData的实例作为参数。

```
static class ExtendMyData        MyData类的实例
{                                     ↓
   public static double Average( MyData md )
   {
      return md.Sum() / 3;
   }          ↑
}        使用MyData的实例


class Program
{
   static void Main()
   {                                                   MyData的实例
      MyData md = new MyData(3, 4, 5);                       ↓
      Console.WriteLine("Average: {0}", ExtendMyData.Average(md));
   }                                                  ↑
}                                                调用静态方法
```

这段代码产生以下输出：

```
Average: 4
```

尽管这是非常好的解决方案，但如果能在类的实例自身上调用该方法，而不是创建另一个作用于它的类的实例，它将会更优雅。下面两行代码阐明了它们的区别。第一行使用刚展示的方法：在另一个类的实例上调用静态方法。第二行展示了我们愿意使用的形式：在对象自身上调用实例方法。扩展方法允许你使用第二种形式，即使第一种形式可能是编写这种调用的正常方法。

```
ExtendMyData.Average( md )                  // 在另一个类的实例上调用静态方法
md.Average();                               // 在对象自身上调用实例方法
```

通过对方法Average的声明做一个小小的改动，就可以使用实例调用形式。需要做的修改是在参数声明中的类型名前增加关键字this，如下面所示。把this关键字加到静态类的静态方法的第一个参数上，把该方法从类ExtendMyData的正规方法改变为类MyData的扩展方法。现在两种调用形式都可以使用。

```
 必须是一个静态类
 ↓
static class ExtendMyData
{    必须是公有的和静态的      关键字和类型
             ↓                     ↓
   public static double Average( this MyData md )
   {
      ...
   }
}
```

扩展方法重要的需求如下：

- 扩展方法必须被声明为static。
- 扩展方法声明所在的类也必须被声明为static。
- 扩展方法必须包含关键字this作为它的第一个参数类型，并在后面跟着它所扩展的类的名称。

图7-21阐明了扩展方法的结构。

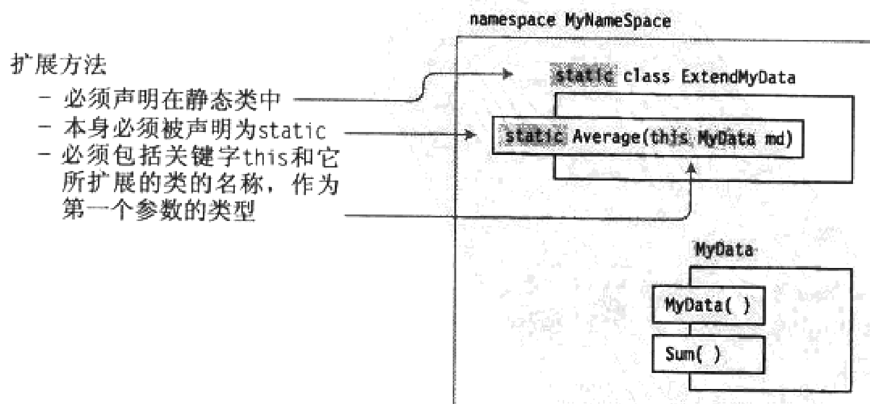



图7-21 扩展方法的结构

下面的代码展示了一个完整的程序，包括类MyData和声明在类ExtendMyData中的扩展方法Average。注意方法Average完全如同它是MyData的实例成员那样被调用！图7-21阐明了这段代码。类MyData和ExtendMyData共同起到期望的类的作用，带有三个方法。

```
namespace ExtensionMethods
{
   sealed class MyData
   {
      private double D1, D2, D3;
      public MyData(double d1, double d2, double d3)
         { D1 = d1; D2 = d2; D3 = d3; }

      public double Sum() { return D1 + D2 + D3; }
   }

   static class ExtendMyData           关键字和类型
   {                                        ↓
      public static double Average(this MyData md)
      {          ↑
           声明为静态的
         return md.Sum() / 3;
      }
   }

   class Program
   {
      static void Main()
      {
         MyData md = new MyData(3, 4, 5);
         Console.WriteLine("Sum:     {0}", md.Sum());
         Console.WriteLine("Average: {0}", md.Average());
      }                                        ↑
   }                                 当作类的实例成员来调用
}
```

这段代码产生以下输出：

```
Sum:     12
Average: 4
```

## 7.14 外部方法

外部方法（external method）是在声明中没有实现的方法。其实现常常是用C#之外的语言编写的。

- 外部方法使用extern修饰符标记，而且在类的声明中没有实现。它的实现被分号取代。

```
  关键字
    ↓
public static extern int GetCurrentDirectory(int size, StringBuilder buf);
                                                                       ↑
                                                                    没有实现
```

- 声明和实现的连接是依赖实现的，但常常使用DllImport特性完成。特性在第24章阐述。

例如，下面的代码使用一个外部方法GetCurrentDirectory，它的实现是Win32系统调用，获取包含当前目录的字符串。

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace ExternalMethod
{
   class MyClass
   {
      [DllImport("kernel32", SetLastError=true)]
      public static extern int GetCurrentDirectory(int a, StringBuilder b);
   }

   class Program
   {
      static void Main( )
      {
         const int MaxDirLength = 250;
         StringBuilder sb = new StringBuilder();
         sb.Length = MaxDirLength;

         MyClass.GetCurrentDirectory(MaxDirLength, sb);
         Console.WriteLine(sb);
      }
   }
}
```

这段代码产生以下输出：

```
C:\BookPrograms\ExternalMethod\ExternalMethod\bin\Debug
```

第 8 章

# 表达式和运算符

**本章内容**

- ❑ 表达式
- ❑ 字面量
- ❑ 求值顺序
- ❑ 简单算术运算符
- ❑ 求余运算符
- ❑ 关系比较运算符和相等比较运算符
- ❑ 递增运算符和递减运算符
- ❑ 条件逻辑运算符
- ❑ 逻辑运算符
- ❑ 移位运算符
- ❑ 赋值运算符
- ❑ 条件运算符
- ❑ 一元算术运算符
- ❑ 用户定义类型转换
- ❑ 运算符重载
- ❑ typeof运算符

## 8.1 表达式

本章将详细说明表达式，并描写C#提供的运算符。本章还将解释如何能定义C#运算符与你的用户定义类一起工作。

表达式是运算符和操作数的字符串。可以担当操作数的结构有：

- ❑ 字面量
- ❑ 常量
- ❑ 变量
- ❑ 方法调用
- ❑ 元素访问器，如数组访问器和索引

- 其他表达式

C#运算符带一个、两个或三个操作数。运算符：

- 把操作数当作输入；
- 执行一个行为；
- 基于行为返回一个值。

表达式可以使用运算符连接以创建其他表达式，如下面的表达式所示，它有三个运算符和四个操作数。

```
a + b
expr  + c
   expr   + d     } a + b + c + d
```

表达式求值是应用每个运算符到它的操作数的过程，以适当的顺序产生一个值。

- 值被返回到表达式被求值的位置。在那里，按次序它可能是一个封装表达式的操作数。
- 除了返回值以外，一些表达式还有副作用，比如在内存中设置一个值。

## 8.2 字面量

字面量（literal）是原代码中书写的数字或字符串，表示一个明确类型的明确的、固定的值。

例如，下面的代码展示了六个类型的字面量。请注意例如double字面量和float字面量的区别。

```
static void Main()          Literals
{
   Console.WriteLine("{0}", 1024);           // 整数字面量
   Console.WriteLine("{0}", 3.1416);         // 双精度型字面量
   Console.WriteLine("{0}", 3.1416F);        // 浮点型字面量
   Console.WriteLine("{0}", true);           // 布尔型字面量
   Console.WriteLine("{0}", 'x');            // 字符型字面量
   Console.WriteLine("{0}", "Hi there");     // 字符串字面量
}
```

这段代码的输出如下：

```
1024
3.1416
3.1416
True
x
Hi there
```

因为字面量是写进源代码的，所以它们的值必须在编译期可知。

个别预定义类型有自己的字面量形式：

- bool有两个字面量：true和false。

❑ 对于引用类型变量，字面量null表示变量没有被设置内存中的引用。

### 8.2.1 整数字面量

整数字面量是最常用的字面量。它们被书写为十进制数字序列，并且：

❑ 没有小数点；

❑ 带有可选的后缀，指明整数的类型。

例如，下面几行展示四个字面量，都是整数236。依据它们的后缀，每个常数都被编译器解释为不同的整数类型。

```
236          // 整型
236L         // 长整型
236U         // 无符号整型
236UL        // 无符号长整型
```

整数类型字面量还可以被写成16进制（hex）形式。数字必须是16进制数（从0到F），而且字符串必须以0x或0X开始（数字*0*，字母*x*）。

整数字面量格式的形式如图8-1所示。方括号内名称的元素是可选的。

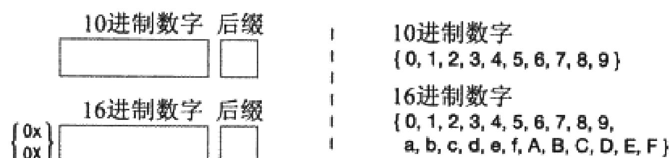



图8-1 整数字面量的格式

整数字面量的后缀见表8-1。对于已知的后缀，编译器会把数字字符串解释为能表示该值而不丢失数据的相应类型中最小的类型。

例如，对于常数236和5000000000，它们都没有后缀。因为236可以用32比特表示，所以它会被编译器解释为一个int。然而较大的数不适合32比特，所以编译器会把它解释为一个long。

表8-1 整数字面量的后缀

| 后　缀 | 整数类型 | 备　注 |
| --- | --- | --- |
| 无 | int、uint、long、ulong | |
| U、u | uint、ulong | |
| L、l | long、ulong | 不推荐使用小写字母*l*，因为它很容易和数字*1*弄错 |
| ul、uL、Ul、UL、lu、Lu、lU、LU | ulong | 不推荐使用小写字母*l*，因为它很容易和数字*1*弄错 |

### 8.2.2 实数字面量

实数字面量的组成如下：

❑ 10进制数字

❑ 一个可选的小数点

- 一个可选的指数部分
- 一个可选的后缀

例如，下面的代码展示了实数类型字面量的不同格式：

```
float  f1 = 236F;
double d1 = 236.714;
double d2 = .35192;
double d3 = 6.338e-26;
```

实数字面量的有效格式如图8-2。方括号内名称的元素是可选的。实数后缀和它们的含义如表8-2所示。

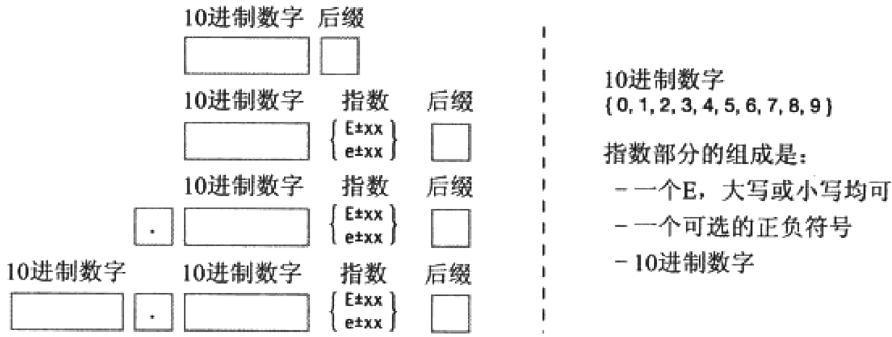



图8-2　实数字面量格式

表8-2　实数字面量的后缀

| 后　　缀 | 实 数 类 型 |
|---|---|
| 无 | double |
| F、f | float |
| D、d | double |
| M、m | decimal |

**说明**　无后缀的实数字面量是double类型，不是float类型！

## 8.2.3　字符字面量

字符字面量由单引号内的字符表示组成。字符字面量可以是下面的任意一种：单个字符、一个简单转义序列、一个十六进制转义序列或一个Unicode转义序列。

- 字符字面量的类型是char。
- 简单转义序列是一个反斜杠后面跟着单个字符。
- 十六进制转义序列是一个反斜杠，后面跟着一个大写或小写的*x*，后面再跟着多至四个十六进制数字。
- Unicode转义序列是一个反斜杠，后面跟着一个大写或小写的*u*，后面再跟着多至四个十六进制数字。

例如，下面的代码展示了字符字面量的不同格式：

```
char c1 = 'd';                    // 单个字符
char c2 = '\n';                   // 简单转义序列
char c3 = '\x0061';               // 十六进制转义序列
char c4 = '\u005a';               // 转义序列
```

一些重要的特殊字符和它们的编码如表8-3所示。

表8-3 重要的特殊字符

| 名 称 | 转 义 序 列 | 十六进制编码 |
|---|---|---|
| 空字符（Null） | \0 | 0x0000 |
| 警告（Alert） | \a | 0x0007 |
| 退格符（Backspace） | \b | 0x0008 |
| 水平制表符（Horizontal Tab） | \t | 0x0009 |
| 换行符 | \n | 0x000A |
| 垂直制表符（Vertial Tab） | \b | 0x000B |
| 换页符 | \f | 0x000C |
| 回车符 | \r | 0x000D |
| 双引号 | \" | 0x0022 |
| 单引号 | \' | 0x0027 |
| 反斜杠 | \\ | 0x005C |

## 8.2.4 字符串字面量

字符串字面量使用双引号标记，而不是字符字面量使用的单引号。有两种字符串字面量类型：

- 规则字符串字面量
- 逐字字符串字面量

规则字符串字面量由双引号内的字符序列组成。规则字符串字面量可以包含：

- 字符
- 简单转义字符
- 十六进制和Unicode转义序列

这里有个示例：

```
string st1 = "Hi there!";
string st2 = "Val1\t5, Val2\t10";
string st3 = "Add\x000ASome\u0007Interest";
```

逐字字符串字面量的书写如同规则字符串字面量，但以一个@字符为前缀。逐字字符串字面量有以下重要特征：

- 逐字字面量与规则字符串字面量区别在于转义字符串不会被求值。在双引号中间的所有东西，包括通常被认为是转义序列的东西，都被严格按字符串中列出的那样打印。

- 逐字字面量的唯一例外是相邻的双引号组，它们被解释为单个双引号字符。

例如，下面的代码比较了规则字符串字面量和逐字字符串字面量：

```
string rst1 = "Hi there!";
string vst1 = @"Hi there!";

string rst2 = "It started, \"Four score and seven...\"";
string vst2 = @"It started, ""Four score and seven...""";

string rst3 = "Value 1 \t 5, Val2 \t 10";
string vst3 = @"Value 1 \t 5, Val2 \t 10";

string rst4 = "C:\\Program Files\\Microsoft\\";
string vst4 = @"C:\Program Files\Microsoft\";

string rst5 = " Print \x000A Multiple \u000A Lines";
string vst5 = @" Print
 Multiple
 Lines";
```

打印这些字符串产生以下输出：

```
Hi there!
Hi there!

It started, "Four score and seven..."
It started, "Four score and seven..."

Value 1          5, Val2         10
Value 1 \t 5, Val2 \t 10

C:\Program Files\Microsoft\
C:\Program Files\Microsoft\

 Print
 Multiple
 Lines

 Print
 Multiple
 Lines
```

**说明**　编译器通过让相同的字符串字面量共享堆中同一内存位置以节约内存。

## 8.3　求值顺序

表达式可以由许多嵌套的子表达式构成。子表达式的求值顺序可以使表达式的最终值发生差别。

例如，已知表达式3*5+2，依照子表达式的求值顺序，有两种可能的结果，如图8-3所示。

❑ 如果乘法先执行，结果是17。

❑ 如果5和2首先加在一起，结果为21。

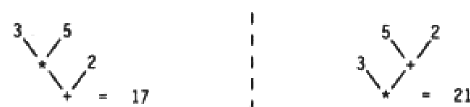



图8-3 简单的求值顺序

## 8.3.1 优先级

从小学学到的知识可以知道，在前面的示例中，乘法必须在加法之前执行，因为乘法比加法有更高的优先级。但和小学学到的知识不同，那时有四个运算符和两个级别的优先级，C#中事情更复杂一些，它有超过45个运算符和14个级别的优先级。

全部的运算符和它们的优先级列表如表8-4所示。该表把最高优先级运算符列在顶端，并持续下降到底端最低优先级运算符。

**表 8-4 运算符优先级：从高到低**

| 分 类 | 运 算 符 |
|---|---|
| 初级运算符 | a.x、f(x)、a[x]、x++、x--、new、typeof、checked、unchecked |
| 一元运算符 | +、-、!、~、++x、--x、(T)x |
| 乘法 | *、/、% |
| 加法 | +、- |
| 移位 | <<、>> |
| 关系和类型 | <、>、<=、>=、is、as |
| 相等 | ==、!= |
| 位与 | & |
| 位异或 | ^ |
| 位或 | \| |
| 条件与 | && |
| 条件或 | \|\| |
| 条件选择 | ?: |
| 赋值运算符 | =、*=、/=、%=、+=、-=、<<=、>>=、&=、^=、\|= |

## 8.3.2 结合性

如果表达式中所有运算符都有不同的优先级，那么计算每个子表达式，从级别最高的开始，按优先等级一直做下去。

但如果两个连续的运算符有相同的优先级别会怎样呢？例如，已知表达式2/6*4，有两个可能的求值顺序：

(2/6)*4=4/3

或

2/(6*4)=1/12

当连续的运算符有相同的优先级时，求值顺序由操作结合性决定。也就是说，已知两个相同优先级的运算符，依照运算符的结合性，其中的一个或另一个优先。运算符的结合性的一些重要特征如下，并且表8-5对此做了总结：

- 左结合运算符从左至右求值。
- 右结合运算符从右至左求值。
- 除赋值运算符以外，二元运算符是左结合的。
- 赋值运算符和条件运算符是右结合的。

因此，已知这些规则，前面的示例表达式应该从左至右分组为(2/6)*4，得到4/3。

表8-5 运算符结合性总结

| 运算符类型 | 结合性 |
|---|---|
| 赋值运算符 | 右结合 |
| 其他二元运算符 | 左结合 |
| 条件运算符 | 右结合 |

可以使用圆括号显式地设定子表达式的求值顺序。括号内的子表达式：

- 覆写优先级和结合性规则。
- 求值顺序从最内嵌套到最外部。

## 8.4 简单算术运算符

简单算术运算符执行基本四则算术运算，如表8-6。这些运算符是二元左结合运算符。

表8-6 简单算术运算符

| 运算符 | 名称 | 描述 |
|---|---|---|
| + | 加 | 计算两个操作数的和 |
| - | 减 | 从第一个操作数中减去第二个操作数 |
| * | 乘 | 求两个操作数的乘积 |
| / | 除 | 用第二个操作数除第一个。整数除法把结果四舍五入到最近的整数 |

算术运算符在所有预定义简单数学类型上执行标准的算术运算。

下面是简单算术运算符的示例：

```
int x1 = 5 + 6;        double d1 = 5.0 + 6.0;
int x2 = 12 - 3;       double d2 = 12.0 - 3.0;
int x3 = 3 * 4;        double d3 = 3.0 * 4.0;
int x4 = 10 / 3;       double d4 = 10.0 / 3.0;

byte b1 = 5 + 6;
sbyte sb1 = 6 * 5;
```

## 8.5 求余运算符

求余运算符（%）用第二个操作数除第一个操作数，忽略掉商，并返回余数。它的描述见表8-7。

求余运算符是二元左结合运算符。

表8-7　求余运算符

| 运算符 | 名称 | 描述 |
| --- | --- | --- |
| % | 求余 | 用第二个操作数除第一个操作数并返回余数 |

下面的内容展示了整数求余运算符的示例：

- 0%3=0，因为0除以3得0余0。
- 1%3=1，因为1除以3得0余1。
- 2%3=2，因为2除以3得0余2。
- 3%3=0，因为3除以3得1余0。
- 4%3=1，因为4除以3得1余1。

求余运算符还可以用于实数以得到实余数。

```
Console.WriteLine("0.0f % 1.5f is {0}", 0.0f % 1.5f);
Console.WriteLine("0.5f % 1.5f is {0}", 0.5f % 1.5f);
Console.WriteLine("1.0f % 1.5f is {0}", 1.0f % 1.5f);
Console.WriteLine("1.5f % 1.5f is {0}", 1.5f % 1.5f);
Console.WriteLine("2.0f % 1.5f is {0}", 2.0f % 1.5f);
Console.WriteLine("2.5f % 1.5f is {0}", 2.5f % 1.5f);
```

这段代码产生以下输出：

```
0.0f % 1.5f is 0          //  0.0 / 1.5 = 0 remainder  0
0.5f % 1.5f is 0.5        //  0.5 / 1.5 = 0 remainder .5
1.0f % 1.5f is 1          //  1.0 / 1.5 = 0 remainder  1
1.5f % 1.5f is 0          //  1.5 / 1.5 = 1 remainder  0
2.0f % 1.5f is 0.5        //  2.0 / 1.5 = 1 remainder .5
2.5f % 1.5f is 1          //  2.5 / 1.5 = 1 remainder  1
```

## 8.6　关系比较运算符和相等比较运算符

关系比较运算符和相等比较运算符是二元运算符，比较它们的操作数并返回bool型值。这些运算符如表8-8所示。

关系运算符和相等比较运算符是二元运算符，并左结合。

表8-8　关系比较运算符和相等比较运算符

| 运算符 | 名称 | 描述 |
| --- | --- | --- |
| < | 小于 | 如果第一个操作数小于第二个操作数，返回true，否则返回false |
| > | 大于 | 如果第一个操作数大于第二个操作数，返回true，否则返回false |
| <= | 小于等于 | 如果第一个操作数小于等于第二个操作数，返回true，否则返回false |
| >= | 大于等于 | 如果第一个操作数大于等于第二个操作数，返回true，否则返回false |
| == | 等于 | 如果第一个操作数等于第二个操作数，返回true，否则返回false |
| != | 不等于 | 如果第一个操作数不等于第二个操作数，返回true，否则返回false |

带有关系或等于运算符的二元表达式返回bool型值。

**说明** 与C和C++不同，在C#中数字不具有布尔意义。

```
int x = 5;
if( x )          // 错，x是int类型，不是布尔类型
   ...
if( x == 5 )     // 对，表达式返回了一个布尔类型的值
   ...
```

打印后，布尔值true和false表示为字符串输出值True和False。

```
int x = 5, y = 4;
Console.WriteLine("x == x is {0}", x == x);
Console.WriteLine("x == y is {0}", x == y);
```

这段代码的输出如下：

```
x == x is True
x == y is False
```

## 比较操作和相等性操作

当比较大多数引用类型的相等性时，只有引用被比较。

- 如果引用相等，也就是说，如果它们指向内存中相同对象，那么相等性比较为true，否则为false，即使内存中两个分离的对象在所有其他方面都完全相等。
- 这称为浅比较。

图8-4阐明了引用类型的比较。

- 在图的左边，a和b两者的引用是相同的，所以比较会返回true。
- 在图的右边，引用不相同，所以即使两个AClass对象的内容完全相同，比较也会返回false。

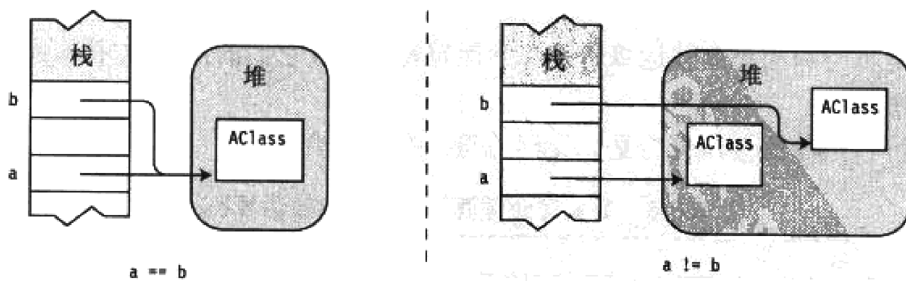



图8-4 比较引用类型的相等性

string类型对象也是引用类型，但它的比较方式不同。比较字符串的相等性时，比较它们的长度和大小写敏感的内容。

- 如果两个字符串有相同的长度和相同大小写敏感的内容，那么相等性比较返回true，即使

它们占用不同的内存区域。

- 这称为深比较（deep comparison）。

委托也是引用类型，并且也使用深比较。委托将在第15章阐述。比较委托的相等性时，如果两个委托都是null，或两者的调用列表中有相同数目的成员，并且调用列表相匹配，那么比较返回true。

比较数值表达式时，类型和值被比较。

比较enum类型时，比较操作数的底层值。枚举将在第13章阐述。

## 8.7 递增运算符和递减运算符

递增运算符对操作数加1。递减运算符从操作数减1。运算符和它们的描述见表8-9。

这些运算符是一元的，并有两种形式：前置形式和后置形式，它们产生不同的效果。

- 在前置形式中，运算符放在操作数之前，例如：++x和--y。
- 在后置形式中，运算符放在操作数之后，例如：x++和y--。

**表8-9 递增运算符和递减运算符**

| 运算符 | 名称 | 描述 |
| --- | --- | --- |
| ++ | 前置递增++Var | 变量的值加1并保存<br>返回变量的新值 |
| | 后置递增Var++ | 变量的值加1并保存<br>返回变量递增之前的旧值 |
| -- | 前置递减--Var | 变量的值减1并保存<br>返回变量的新值 |
| | 后置递减Var-- | 变量的值减1并保存<br>返回变量递减之前的旧值 |

比较运算符的前置和后置形式。

- 无论运算符使用前置形式还是后置形式，在语句执行之后，最终存放在操作数的变量中的值是相同的。
- 唯一不同的是运算符返回给表达式的值。

表8-10中展示的示例总结了运算符行为。

**表8-10 前置和后置的递增和递减运算符的行为**

| | 表达式：x=10 | 返回给表达式的值 | 计算后变量的值 |
| --- | --- | --- | --- |
| 前置递增 | ++x | 11 | 11 |
| 后置递增 | x++ | 10 | 11 |
| 前置递减 | --x | 9 | 9 |
| 后置递减 | x-- | 10 | 9 |

例如，下面是四个不同版本运算符的一个简单示范。为了展示相同输入情况下的不同结果，

操作数x在每个赋值语句之前被重新设定为5。

```
int x = 5, y;
y = x++;   // result: y: 5, x: 6
Console.WriteLine("y: {0}, x: {1}", y, x);

x = 5;
y = ++x;   // result: y: 6, x: 6
Console.WriteLine("y: {0}, x: {1}", y, x);

x = 5;
y = x--;   // result: y: 5, x: 4
Console.WriteLine("y: {0}, x: {1}", y, x);

x = 5;
y = --x;   // result: y: 4, x: 4
Console.WriteLine("y: {0}, x: {1}", y, x);
```

这段代码产生以下输出：

```
y: 5, x: 6
y: 6, x: 6
y: 5, x: 4
y: 4, x: 4
```

## 8.8 条件逻辑运算符

逻辑运算符用于比较或否定它们的操作数的逻辑值，并返回结果逻辑值。运算符如表8-11所示。

逻辑与（AND）和逻辑或（OR）运算符是二元左结合运算符。逻辑非（NOT）是一元运算符。

表8-11　条件逻辑运算符

| 运　算　符 | 名　称 | 描　述 |
|---|---|---|
| && | 与 | 如果两个操作数都是true，结果为true；否则为false |
| \|\| | 或 | 如果至少一个操作数是true，结果为true；否则为false |
| ! | 非 | 如果操作数是false，结果为true；否则为false |

这些运算符的语法如下，其中Exp1和Exp2计算成布尔值：

```
Expr1 && Expr2
Expr1 || Expr2
   !Expr
```

下面是一些示例：

```
bool bVal;
```

```
bVal = (1 == 1) && (2 == 2);       // True, 两个表达式同为真
bVal = (1 == 1) && (1 == 2);       // False, 第二个表达式为假

bVal = (1 == 1) || (2 == 2);       // True, 两个表达式同为真
bVal = (1 == 1) || (1 == 2);       // True, 第一个表达式为真
bVal = (1 == 2) || (2 == 3);       // False, 两个表达式同为假

bVal = true;                       // 设置bVal为真
bVal = !bVal;                      // bVal为假
```

条件逻辑运算符使用“短路（short circuit）”模式操作，意思是，如果计算*Expr1*之后结果已经确定了，那么它会跳过*Expr2*的求值。下面的代码展示了表达式的示例，在表达式中，计算第一个操作数之后值就能被确定：

```
bool bVal;
bVal = (1 == 2) && (2 == 2);     // False, 因为后计算前面的表达式

bVal = (1 == 1) || (1 == 2);     // True, 因为后计算前面的表达式
```

由于这种短路行为，不要在*Expr2*中放置带副作用的表达式（比如改变一个值），因为它可能不被求值。在下面的代码中，变量iVal的后置递增不会被执行，因为执行了第一个子表达式之后，就可以确定整个表达式的值是false。

```
bool bVal; int iVal = 10;

bVal = (1 == 2) && (9 == iVal++);        //结果: bVal = False, iVal = 10
             ↑              ↑
         不会被计算        错误
```

## 8.9 逻辑运算符

按位逻辑运算符常用于设置比特形式的方法参数。按位逻辑运算符如表8-12所示。

这些运算符，除按位非以外，都是二元左结合运算符，按位非是一元运算符。

表8-12 逻辑运算符

| 运算符 | 名称 | 描述 |
|---|---|---|
| & | 位与 | 产生两个操作数的按位与。仅当两个操作位都为1时结果位才是1 |
| \| | 位或 | 产生两个操作数的按位或。只要任意一个操作位为1结果位就是1 |
| ^ | 位异或 | 产生两个操作数的按位异或。仅当一个而不是两个操作位为1时结果位为1 |
| ~ | 位非 | 操作数的每个位都取反。该操作得到操作数的二进制反码 |

二元按位运算符比较它的两个操作数中每个位置的相应位，并依据逻辑操作设置返回值中的位。

图8-5展示了四个按位逻辑操作的示例。

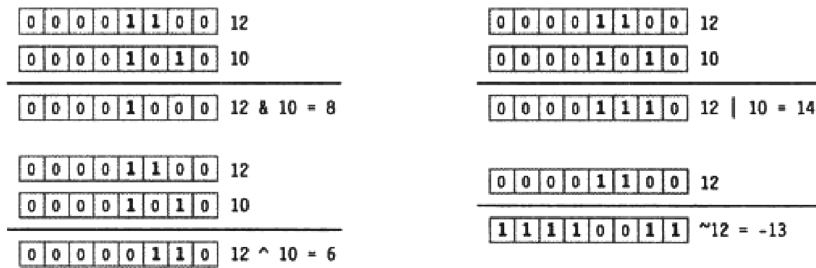



图8-5　按位逻辑操作示例

下面的代码实现了前面的示例：

```
const byte x = 12, y = 10;
sbyte a;

a = x & y;                    //  a = 8
a = x | y;                    //  a = 14
a = x ^ y;                    //  a = 6
a = ~x;                       //  a = -13
```

## 8.10　移位运算符

按位移位运算符向左或向右把位组移动指定数量个位置，空出的位用0填充。移位运算符如表8-13所示。

移位运算符是二元左结合运算符。按位移位运算符如下面所示。移动的位置数由*Count*给出。

```
Operand << Count                    // Left shift
Operand >> Count                    // Right shift
```

表8-13　移位运算符

| 运算符 | 名　称 | 描　　述 |
|---|---|---|
| << | 左移 | 向左移动位组已知数目个位置。位从左边移出并丢失。右边打开的比特位置被用0填充 |
| >> | 右移 | 向右移动位组已知数目个位置。位从右边移出并丢失 |

对于绝大多数的C#编程，你无需了解关于硬件底层的任何事情。然而，如果你在做有符号数字的位操作，了解数值的表示法会有帮助。底层的硬件使用一种名称为**二进制补码**的形式表示有符号二进制数。在二进制补码表示法中，正数使用正常的二进制形式。要取一个数的相反数，把这个数按位取反再加1。这个过程把一个正数转换成它的负数的表示法，反之亦然。在二进制补码中，所有的负数最左边的比特位置都是1。图8-6展示了12的相反数。

在移位有符号数时，底层表示法就很重要，因为左移一个整数一位的结果与把它乘以2的结果相同。把它右移一位的结果和除以2相同。

然而，如果你右移一个负数，最左边的位被用0填充，这将会产生一个错误的结果。最左边位置的0标志一个正数。但这是不正确的，因为一个负数除以2不能得到一个正数。

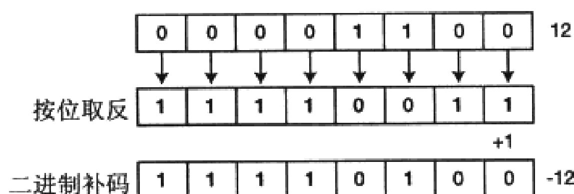



图8-6 要获取二进制补码的相反数，把它按位取反再加1

为了适合这种情形，当操作数是有符号整数时，如果操作数最左边的比特位是1（标志一个负数），在左边移开的比特位置被用1而不是0填充。这维持了正确的二进制补码表示法。对于正数或无符号数，左边移开的比特位置用0填充。

图8-7展示了表达式14<<3在一个byte中是如何求值的。该操作导致：

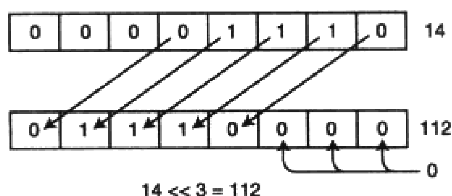



图8-7 左移三位的示例

- 操作数（14）的每个位都向左移动了三个位置。
- 右边结尾腾出的位置被用0填充。
- 结果值是112。

图8-8阐明了位移动操作。




图8-8 移位

下面的代码实现了前面的示例：

```
int a, b, x = 14;

a = x << 3;                // 左移
b = x >> 3;                // 右移

Console.WriteLine("{0} << 3 = {1}", x, a);
Console.WriteLine("{0} >> 3 = {1}", x, b);
```

这段代码产生以下输出：

```
14 << 3 = 112
14 >> 3 = 1
```

## 8.11　赋值运算符

赋值运算符对运算符右边的表达式求值，并用该值设置运算符左边的变量表达式。赋值运算符如表8-14所示。

赋值运算符是二元左结合运算符。

表8-14　赋值运算符

| 运　算　符 | 描　　述 |
|---|---|
| = | 简单赋值，计算右边表达式的值，并把返回值赋给左边的变量或表达式 |
| *= | 复合赋值，var*=expr等价于var = var*(expr) |
| /= | 复合赋值，var/=expr等价于var = var/(expr) |
| %= | 复合赋值，var%=expr等价于var = var%(expr) |
| += | 复合赋值，var+=expr等价于var = var+(expr) |
| -= | 复合赋值，var-=expr等价于var = var-(expr) |
| <<= | 复合赋值，var<<=expr等价于var = var<<(expr) |
| >>= | 复合赋值，var>>=expr等价于var = var>>(expr) |
| &= | 复合赋值，var&=expr等价于var = var&(expr) |
| ^= | 复合赋值，var^=expr等价于var = var^(expr) |
| \|= | 复合赋值，va\|=expr等价于var = var\|(expr) |

语法如下：

```
VariableExpression Operator Expression
```

对于简单赋值，运算符右边的表达式被求值，而且它的值被赋给左边的变量。

```
int x;
x = 5;
x = y * z;
```

可以放在赋值运算符左边的对象类型如下。它们将在后面的内容中讨论。

- 变量（本地变量、字段、参数）
- 属性
- 索引
- 事件

### 复合赋值

经常地，你会想求一个表达式的值，并把结果加到一个变量的当前值上，如下所示：

```
x = x + expr;
```

复合赋值运算符允许一种速记方法，在某些普通情况下避免左边的变量在右边重复出现。例如，下面两条语句是语义等价的，但第二条更短一些，而且确实易于理解。

```
x = x + (y - z);
x += y - z;
```

其他的复合赋值语句与之类似：

```
                        注意括号
                          ↓     ↓
x *= y - z;      // 等价于 x = x * (y - z)
x /= y - z;      // 等价于 x = x / (y - z)
  ...
```

## 8.12 条件运算符

条件运算符是一种强大且简洁的方法，基于条件的结果，返回两个值之一。运算符如表8-15所示。

条件运算符是三元运算符。

表 8-15 条件运算符

| 运算符 | 名称 | 描述 |
| --- | --- | --- |
| ?: | 条件运算符 | 对一个表达式求值，并依据表达式是否返回 true 或 false，返回两个值之一 |

条件运算符的语法如下所示。它有一个测试表达式和两个结果表达式。

- *Condition*必须返回一个bool型值。
- 如果Condition求值为true，那么Expression1被求值并返回。否则，Expression2被求值并返回。

```
Condition ? Expression1 : Expression2
```

条件运算符可以和if…else结构相比。例如，下面的if…else结构检查一个条件，如果条件为真，把5赋值给变量intVar，否则给它赋值10。

```
if ( x < y )                                     // if...else
   intVar = 5;
else
   intVar = 10;
```

条件运算符能以较少冗长的方式实现相同的操作，如下面的语句所示：

```
intVar = x < y  ?  5  :  10;                     // 条件运算符
```

把条件和每个返回值放在分离的行，如下面的代码所示，使操作意图非常容易理解。

```
intVar = x < y
         ?  5
         :  10 ;
```

图8-9比较了示例中所示的两种形式

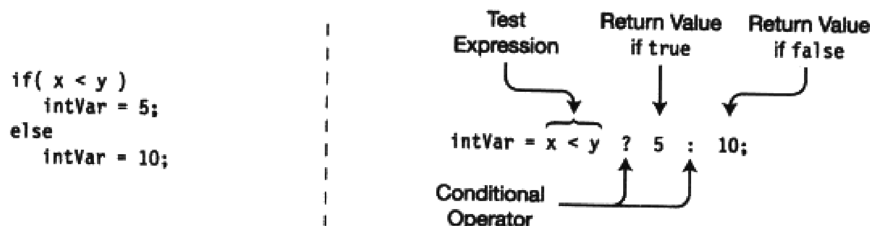



图8-9　条件运算符对比if…else

例如，下面的代码使用条件运算符三次，每个WriteLine语句一次。在第一个示例中，它返回x的值或y的值。在另两个示例中，它返回空字符串或字符串“not”。

```
int x = 10, y = 9;
int highVal = x > y                          // 条件
                 ? x                         // 表达式 1
                 : y;                        // 表达式 2
Console.WriteLine("highVal:  {0}\n", highVal);

Console.WriteLine("x is{0} greater than y",
                     x > y                   // 条件
                     ? ""                    // 表达式 1
                     : " not");              // 表达式 2
y = 11;
Console.WriteLine("x is{0} greater than y",
                     x > y                   // 条件
                     ? ""                    // 表达式 1
                     : " not");              // 表达式 2
```

这段代码产生以下输出：

```
highVal:  10

x is greater than y
x is not greater than y
```

**说明**　if…else语句是控制流语句。它应当被用于做两个动作中的一个或另一个。条件运算符返回一个表达式。它应当被用于返回两个值中的一个或另一个。

## 8.13　一元算术运算符

一元算术运算符设置数字值的符号，如表8-16所示。

- 一元正运算符简单返回操作数的值。
- 一元负运算符返回0减操作数得到的值。

表8-16 一元运算符

| 运 算 符 | 名 称 | 描 述 |
|---|---|---|
| + | 正号 | 返回操作数的数值 |
| - | 负号 | 返回0减操作数得到的值 |

例如，下面的代码展示了运算符的使用和结果：

```
int x = 10;         // x = 10
int y = -x;         // y = -10
int z = -y;         // z = 10
```

## 8.14 用户定义类型转换

用户定义转换在第18章有非常详细讨论，但在这里也将提到它们，因为它们是运算符。

- 可以为自己的类和结构定义隐式转换和显式转换。这允许把用户定义类型的对象转换成某个其他类型，反之亦然。
- C#提供隐式转换和显式转换。
  - 对于**隐式转换**，当在决定什么类型用于特定上下文时，如有必要，编译器会自动执行转换。
  - 对于**显式转换**，编译器只在使用显式转换运算符时才执行转换。

声明隐式转换的语法如下。public和static修饰符是所有用户定义转换必需的。

```
          必需的                    目标类型          源数据
public static implicit operator TargetType ( SourceType Identifier )
{
    ...
    return ObjectOfTargetType;
}
```

显式转换的语法与之相同，除了explicit替换掉implicit。

下面的代码展示了声明转换运算符的示例，它把类型LimitedInt的对象转换为int型，并反之亦然。

```
class LimitedInt
{                                    目标类型   源数据
   public static implicit operator int (LimitedInt li)
   {
      return li.TheValue;
   }                                      目标类型   源数据
   public static implicit operator LimitedInt (int x)
   {
      LimitedInt li = new LimitedInt();
      li.TheValue = x;
```

```
        return li;
    }

    private int _TheValue = 0;
    public int TheValue{ ... }
}
```

例如，下面的代码重申（reiterate）并使用了刚才定义的两个类型转换。在Main中，一个int字面量被转换为LimitedInt对象，在下一行，LimitedInt被转换成一个int。

```
class LimitedInt
{
    const int MaxValue = 100;
    const int MinValue = 0;

    public static implicit operator int(LimitedInt li)        // 类型转换
    {
        return li.TheValue;
    }

    public static implicit operator LimitedInt(int x)         // 类型转换
    {
        LimitedInt li = new LimitedInt();
        li.TheValue = x;
        return li;
    }

    private int _TheValue = 0;
    public int TheValue                                        // 属性
    {
        get { return _TheValue; }
        set
        {
            if (value < MinValue)
                _TheValue = 0;
            else
                _TheValue = value > MaxValue
                                ? MaxValue
                                : value;
        }
    }
}

class Program
{
    static void Main()                                // Main
    {
        LimitedInt li = 5;                            // 将5转换为LimitedInt
        int Five = li;                                // 将LimitedInt转换为int
```

```
        Console.WriteLine("li: {0}, Five: {1}", li.TheValue, Five);
    }
}
```

### 显式转换和强制转换运算符

前面的示例代码展示了int到LimitedInt类型的隐式转换和LimitedInt类型到int的隐式转换。然而，如果你把两个转换运算符声明为explicit，你将不得不在实行转换时显式使用转换运算符。

强制转换运算符由想要把表达式转换成的类型的名称组成，在一对圆括号内部。例如，在下面的代码中，方法Main把值5强制转换成一个LimitedInt对象。

```
                    强制类型转换操作符
                           ↓
LimitedInt li = (LimitedInt) 5;
```

例如，这里有相应的部分代码，改动的部分被标记出来：

```
                 ↓
public static explicit operator int(LimitedInt li)
{
    return li.TheValue;
}
                 ↓
public static explicit operator LimitedInt(int x)
{
    LimitedInt li = new LimitedInt();
    li.TheValue = x;
    return li;
}

static void Main()
{                         ↓
    LimitedInt li = (LimitedInt) 5;
    int Five = (int) li;
                 ↑
    Console.WriteLine(" li: {0}, Five: {1}", li.TheValue, Five);
}
```

在代码的两个版本中，输出如下：

```
li: 5, Five: 5
```

另外有两个运算符，接受一种类型的值，并返回另一种不同的、指定类型的值。这就是is运算符和as运算符。它们将在第18章结尾阐述。

## 8.15 运算符重载

如你所见，C#运算符被定义为使用预定义类型作为操作数来工作。如果面对一个用户定义类

型，运算符完全不知道如何处理它。运算符重载允许你定义C#运算符应该如何操作自定义类型的操作数。

- 运算符重载只能用于类和结构。
- 为类或结构重载一个运算符x，可以声明一个名称为operator x的方法并实现它的行为（例如：operator +和operator -等）。
  - 一元运算符的重载方法带一个单独的class或struct类型的参数。
  - 二元运算符的重载方法带两个参数，其中至少有一个必须是class或struct类型。

```
public static LimitedInt operator -(LimitedInt x)              // Unary
public static LimitedInt operator +(LimitedInt x, double y)    // Binary
```

运算符重载方法必须被声明为：

- 带static和public两个修饰符。
- 类或结构的成员，该类或结构是它的一个操作数。

例如，下面的代码展示了类LimitedInt的两个重载的运算符：加运算符和减运算符。你可以说它是负数而不是减法，因为运算符重载方法只有一个单独的参数，因此是一元的，而减法运算符是二元的。

```
class LimitedInt Return
{       必需的        类型    关键字  操作符          操作数
           ↓            ↓        ↓       ↓               ↓
   public static LimitedInt operator + (LimitedInt x, double y)
   {
      LimitedInt li = new LimitedInt();
      li.TheValue = x.TheValue + (int)y;
      return li;
   }

   public static LimitedInt operator - (LimitedInt x)
   {
      // In this strange class, negating a value just sets its value to 0.
      LimitedInt li = new LimitedInt();
      li.TheValue = 0;
      return li;
   }
   ...
}
```

### 8.15.1　运算符重载的限制

不是所有运算符都能被重载，可以重载的类型也有限制。关于运算符重载的限制需要了解的重要事情在本节的后面描述。

只有下面这些运算符可以被重载。列表中明显缺少的是赋值运算符。

可重载的一元运算符：+、-、！、~、++、--、true、false

可重载的二元运算符：+、-、*、/、%、&、|、^、<<、>>、==、!=、>、<、>=、<=

递增和递减运算符是可重载的。但和预定义转换不同，重载运算符的前置和后置之间没有区别。

运算符重载不能做下面的事情：

- 创建新运算符。
- 改变运算符的语法。
- 重新定义运算符如何处理预定义类型。
- 改变运算符的优先级或结合性。

---

**说明** 重载运算符应该符合运算符的直观含义。

---

### 8.15.2 运算符重载的示例

下面的代码展示了类LimitedInt的三个运算符的重载：负数、减法和加法。

```
class LimitedInt {
   const int MaxValue = 100;
   const int MinValue = 0;

   public static LimitedInt operator -(LimitedInt x)
   {
      // In this strange class, negating a value just sets its value to 0.
      LimitedInt li = new LimitedInt();
      li.TheValue = 0;
      return li;
   }

   public static LimitedInt operator -(LimitedInt x, LimitedInt y)
   {
      LimitedInt li = new LimitedInt();
      li.TheValue = x.TheValue - y.TheValue;
      return li;
   }

   public static LimitedInt operator +(LimitedInt x, double y)
   {
      LimitedInt li = new LimitedInt();
      li.TheValue = x.TheValue + (int)y;
      return li;
   }

   private int _TheValue = 0;
   public int TheValue
   {
      get { return _TheValue; }
      set
      {
```

```
            if (value < MinValue)
                _TheValue = 0;
            else
                _TheValue = value > MaxValue
                                ? MaxValue
                                : value;
        }
    }
}

class Program {
    static void Main() {
        LimitedInt li1 = new LimitedInt();
        LimitedInt li2 = new LimitedInt();
        LimitedInt li3 = new LimitedInt();
        li1.TheValue = 10; li2.TheValue = 26;
        Console.WriteLine(" li1: {0}, li2: {1}", li1.TheValue, li2.TheValue);

        li3 = -li1;
        Console.WriteLine("-{0} = {1}", li1.TheValue, li3.TheValue);

        li3 = li2 - li1;
        Console.WriteLine(" {0} - {1} = {2}",
                  li2.TheValue, li1.TheValue, li3.TheValue);

        li3 = li1 - li2;
        Console.WriteLine(" {0} - {1} = {2}",
                  li1.TheValue, li2.TheValue, li3.TheValue);
    }
}
```

这段代码产生以下输出：

```
 li1: 10, li2: 26
-10 = 0
 26 - 10 = 16
 10 - 26 = 0
```

## 8.16　typeof 运算符

typeof运算符返回作为它的参数的任何类型的System.Type对象。通过这个对象，可以得到类型的特征。（对任何已知类型，只有一个System.Type对象。）运算符的特征如表8-17所示。

typeof运算符是一元运算符。

表8-17　typeof运算符

| 运　算　符 | 描　　述 |
|---|---|
| typeof | 返回已知类型的System.Type对象 |

下面是typeof运算符语法的示例。Type是System命名空间中的一个类。

```
Type t = typeof ( SomeClass )
```

不能重载typeof运算符，因为这会使.NET的类型安全机制失效。

例如，下面的代码使用typeof运算符以获取名称为SomeClass的类的信息，并打印出它的公有字段和方法的名称。

```
using System.Reflection;

class SomeClass
{
   public int  Field1;
   public int  Field2;

   public void Method1() { }
   public int  Method2() { return 1; }
}

class Program
{
   static void Main()
   {
      Type t = typeof(SomeClass);
      FieldInfo[]  fi = t.GetFields();
      MethodInfo[] mi = t.GetMethods();

      foreach (FieldInfo f in fi)
         Console.WriteLine("Field : {0}", f.Name);
      foreach (MethodInfo m in mi)
         Console.WriteLine("Method: {0}", m.Name);
   }
}
```

这段代码的输出如下：

```
Field : Field1
Field : Field2
Method: Method1
Method: Method2
Method: GetType
Method: ToString
Method: Equals
Method: GetHashCode
```

typeof运算符还被GetType方法调用，该方法对每个类型的每个对象都有效。例如，下面的代码获取对象类型的名称：

```
class SomeClass
```

```
{
   ...
}

class Program
{
   static void Main()
   {
      SomeClass s = new SomeClass();

      Console.WriteLine("Type s: {0}", s.GetType().Name);
   }
}
```

这段代码产生以下输出：

```
Type s: SomeClass
```

# 第 9 章 语　　句

**本章内容**

- 什么是语句
- 表达式语句
- 控制流语句
- if语句
- if…else语句
- switch语句
- while循环
- do循环
- for循环
- 跳转语句
- 标签语句
- goto语句
- using语句
- 其他语句

## 9.1 什么是语句

C#中的语句非常类似于C和C++的语句。这一章阐述C#语句的特征，还有语言提供的控制流语句。

- 语句是描述一个类型或告诉程序去执行一个动作的源代码指令。
- 有三种主要语句种类，如下所示。
  - **声明语句**：声明类型或变量的语句。
  - **嵌入语句**：执行动作或管理控制流的语句。
  - **标签语句**：控制可以跳转到的语句。

前面的章节中阐述了许多不同的声明语句，包括本地变量声明、类声明以及类成员的声明。这一章将阐述嵌入语句，它不声明类型、变量或实例。相反，它们使用表达式和控制流结构与已经被声明语句声明了的对象和变量一起工作。

- 简单语句由一个表达式和后面跟着的分号组成。
- 块是匹配的大括号括起来的语句序列。括起来的语句可以包括：
  - 声明语句
  - 嵌入语句
  - 标签语句
  - 嵌套块

下面的代码给出了每种语句的示例：

```
int x = 10;               // 简单声明
int z;                    // 简单声明

{                         // 块
   int y = 20;            // 简单声明
   z = x + y;             // 嵌入语句
top: y = 30;              // 标签语句
      ...
   {                      // 嵌套块
      ...
   }
}
```

**说明**　块在语法上算作一个单条嵌入语句。任何语法上需要一个嵌入语句的地方，都可以使用块。

空语句仅由一个分号组成。可以把空语句用在任何语言的语法需要一条嵌入语句而程序逻辑又不需要任何动作的地方。

例如，下面的代码是一个使用空语句的示例：

- 代码的第二行是一条空语句。之所以需要它是因为在该结构的if部分和else部分之间必须有一条嵌入语句。
- 第四行是一条简单语句，如结束分号旁边所示。

```
if( x < y )
   ;                     // 空语句
else
   z = a + b;            // 简单语句
```

## 9.2　表达式语句

上一章阐述了表达式。表达式返回值，但它们也有副作用（side effect）。

- 副作用是一种影响程序状态的行为。
- 许多表达式被求值只是为了它们的副作用。

可以在表达式后面放置语句终结符（分号）从一个表达式创建一条语句。表达式返回的任何值都被丢弃。例如，下面的代码展示了一个表达式语句。它由赋值表达式（一个赋值运算符和两个操作数）和后面跟着的一个分号组成。它做下面两件事：

- 该表达式把运行符右边的值赋给变量x引用的内存位置。虽然这可能是这条语句的主要动机，但这被视为副作用。
- 设置了x的值之后，表达式返回x的新值。但没有什么东西接收这个返回值，所以它被忽略了。

```
x = 10;
```

计算这个表达式的全部原因就是完成这个副作用。

## 9.3 控制流语句

C#提供与现代编程语言共同的控制流结构。

- 条件执行依据一个条件执行或跳过一个代码片段。条件执行语句如下：
  - if
  - if…else
  - switch
- 循环语句重复执行一个代码片段。循环语句如下：
  - while
  - do
  - for
  - foreach
- 跳转语句把控制流从一个代码片段改变到另一个代码片段中的指定语句。跳转语句如下：
  - break
  - continue
  - return
  - goto
  - throw

条件执行和循环结构（除了foreach）需要一个测试表达式或条件以决定程序应当在哪里继续执行。

---

**说明** 与C和C++不同，测试表达式必须返回bool型值。数字在C#中没有布尔意义。

---

## 9.4 if 语句

if语句实现按条件执行。if语句的语法如下所示，图9-1阐明了它。

- *TestExpr*必须计算成bool型值。

- 如果*TestExpr*求值为true，*Statement*被执行。
- 如果它求值为false，*Statement*被跳过。

```
if( TestExpr )
   Statement
```

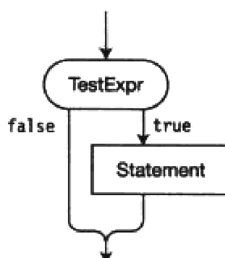



图9-1　if语句

下面的代码展示了if语句的示例：

```
// With a simple statement
if( x <= 10 )
    z = x - 1;                  // 简单语句不需要大括号


// With a block
if( x >= 20 )
{
    x = x - 5;                 // 块需要大括号
    y = x + z;
}


int x = 5;
if( x )               // 错：表达式应该是bool型，而不是int型
{
    ...
}
```

## 9.5　if...else语句

if...else语句实现双路分支。if...else语句的语法如下，图9-2阐明了该语法。

- 如果*TestExpr*求值为true，*Statement1*被执行。
- 如果它求值为flase，换成*Statement2*被执行。

```
If( TestExpr )
   Statement1
else
   Statement2
```

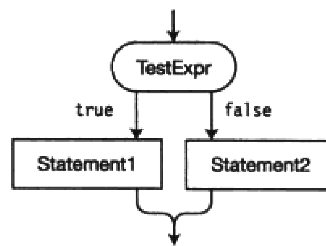



图9-2 if…else语句

下面是if…else语句的示例：

```
If( x <= 10 )
   z = x - 1;                 // 简单语句
else
{                             // 多条语句组成的语句块
   x = x - 5;
   y = x + z;
}
```

## 9.6 switch 语句

switch语句实现多路分支。switch语句的语法和结构如图9-3所示。

- switch语句包含0个或多个分支。
- 每个分支以一个或多个分支标签开始。

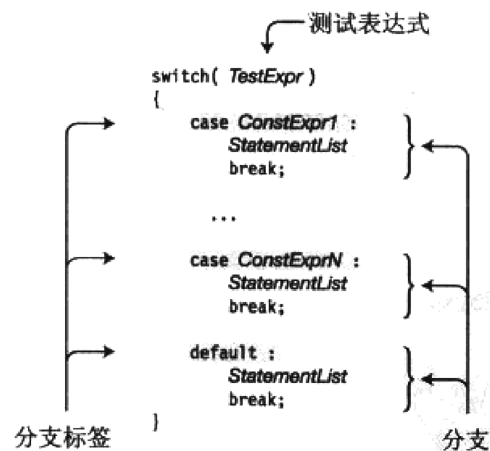



图9-3 switch语句的结构

通过图9-3中结构的控制流如下：

```
case ConstantExpression :
 ↑                      ↑
关键字              分支标签结束符
```

- 测试表达式*TestExpr*在结构的顶端被求值。

- 如果*TestExpr*的值等于值*ConstExpr1*，也就是等于第一个分支标签中的常数表达式，那么跟在分支标签后的语句列表中的语句被执行，直到遇到break语句。
- 每个分支必须以break语句结尾（或goto语句，稍后会讨论）。
- break语句把执行引向switch语句的结尾。
- default段是可选的，但如果包括了，就必须包括一条break语句。

图9-4中阐明了穿过switch语句的一般控制流。可以用goto语句或return语句改变穿过switch语句的控制流。

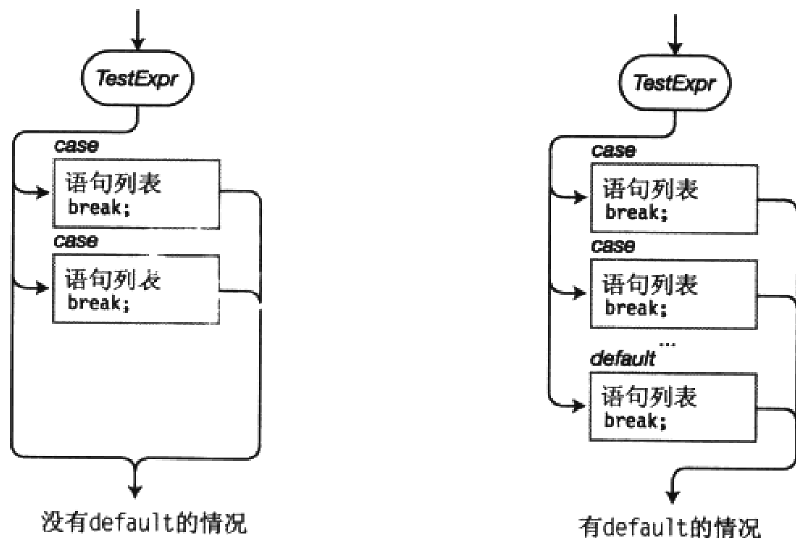



图9-4　穿过switch语句的控制流

**说明**　与C和C++不同，每个分支，包括可选的default段，必须以break或return语句结束。在C#中，没有从一个分支到下一个分支的倒向。

## 9.6.1　分支示例

下面的代码执行switch语句5次，x的值从1变化到5。从输出中可以看出，在循环的每一周期哪个case段被执行。

```
for( int x=1; x<6; x++ )
{
   switch( x )                          // 计算变量 x 的值
   {
      case 2:                                    // 如果 x=2
         Console.WriteLine
            ("x is {0} -- In Case 2", x);
         break;                                  // 结束 switch 语句

      case 5:                                    // 如果 x=5
         Console.WriteLine
```

```
            Console.WriteLine
               ("x is {0} -- In Case 5", x);
            break;                                      // 结束 switch 语句

        default:
                                              // 如果 x 既不等于 2 也不等于 5
            Console.WriteLine
               ("x is {0} -- In Default case", x);
            break;
                                              // 结束 switch 语句
    }
}
```

这段代码产生以下输出：

```
x is 1 -- In Default case
x is 2 -- In Case 2
x is 3 -- In Default case
x is 4 -- In Default case
x is 5 -- In Case 5
```

### 9.6.2 switch 语句的补充

一个switch语句可以有任意数目的分支，包括没有分支（尽管没有分支的时候会得到编译警告）。default段不是必需的，如下面的示例所示。然而，包括default段通常被认为是好的习惯，因为它可以捕获潜在的错误。

例如，下面代码中的switch语句没有default段。该switch语句在一个for循环内部，该循环执行switch语句5次，x的值从1开始到5结束。

```
for( int x=1; x<6; x++ )
{
   switch( x )
   {
      case 5:
         Console.WriteLine("x is {0} -- In Case 5", x);
         break;
   }
}
```

这段代码产生以下输出：

```
x is 5 -- In Case 5
```

下面的代码只有default段：

```
for( int x=1; x<4; x++ )
{
   switch( x )
```

```
    {
      default:
        Console.WriteLine("x is {0} -- In Default case", x);
        break;
    }
  }
```

这段代码产生以下输出：

```
x is 1 -- In Default case
x is 2 -- In Default case
x is 3 -- In Default case
```

### 9.6.3　分支标签

分支标签中跟在关键字case之后的表达式：

- ❑ 必须是常数表达式，而且因此必须在编译期被编译器完全求值。
- ❑ 必须与测试表达式类型相同。

例如，图9-5展示了三个示例switch语句。

```
const string YES = "yes";

string s = "no";
switch (s)
{
   case YES:
      PrintOut("Yes");
      break;

   case "no":
      PrintOut("No");
      break;
}
```

```
const char LetterB = 'b';

char c = 'a';
switch (c)
{
   case 'a':
      PrintOut("a");
      break;

   case LetterB:
      PrintOut("b");
      break;
}
```

```
const int Five = 5;

int x = 5;
switch (x)
{
   case Five:
      PrintOut("5");
      break;

   case 10:
      PrintOut("10");
      break;
}
```

图9-5　带不同类型分支标签的switch语句

尽管C#不允许从一个分支到另一个分支的倒向。

- ❑ 可以把多个分支标签附加到任意分支。
- ❑ 跟在和一个分支关联的语句列表后面，在下一个标签之前，必须是一个break或goto语句，除非在这两个分支标签之间没有插入可执行语句。

例如，在下面的代码中，因为在开始的三个分支标签之间没有可执行语句，它们可以一个接着一个。然而，分支5和6之间有一条可执行语句，所以在分支6之前必须有一个break或goto语句。

```
switch( x )
{
   case 1:
   case 2:
   case 3:
      ...
      break;
   case 5:
```

```
    y = x + 1;
case 6:
    ...
```

## 9.7 while 循环

while循环是一种简单循环结构，其中测试表达式在循环的顶部执行。while循环的语法如下所示，图9-6阐明了它。

- 首先，*TestExpr*被求值。
- 如果*TestExpr*求值为false，那么执行在while循环结尾之后继续。
- 否则，当*TestExpr*求值为true时，*Statement*被执行，并且*TestExpr*再次被求值。每次*TestExpr*求值为true时，*Statement*都要再执行一次。循环在*TestExpr*求值为false时结束。

```
while( TestExpr )
   Statement
```

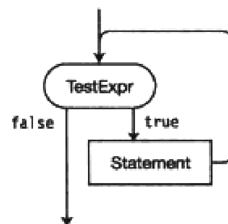



图9-6 while循环

下面的代码展示了一个while循环的示例，其中测试表达式变量从值3开始在每次迭代中递减。当变量的值变为0时循环退出。

```
int x = 3;
while( x > 0 )
{
   Console.WriteLine("x:  {0}", x);
   x--;
}
Console.WriteLine("Out of loop");
```

这段代码产生以下输出：

```
x:  3
x:  2
x:  1
Out of loop
```

## 9.8 do 循环

do循环是一种简单循环结构，其中测试表达式在循环的底部执行。do循环的语法如下所示，

图9-7阐明了它。

- 首先，*Statement*被执行。
- 然后，*TestExpr*被求值。
- 如果*TestExpr*返回true，那么*Statement*再次执行。
- 每次*TestExpr*返回true，*Statement*都执行一次。
- 当*TestExpr*返回false时，控制传递到跟在循环结构结尾之后的那条语句。

```
do
   Statement
while( TestExpr );          // End of do loop
```

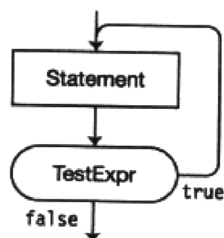



图9-7　do循环

do循环有几个特征，使它与其他控制流结构分开。它们如下：

- 循环体*Statement*总是至少被执行一次，即使*TestExpr*初始为false。
- 在测试表达式的关闭圆括号之后需要一个分号。

下面的代码展示了一个do循环的示例：

```
int x = 0;
do
   Console.WriteLine("x is {0}", x++);
while (x<3);
         ↑
       必需的
```

这段代码产生以下输出：

```
x is 0
x is 1
x is 2
```

## 9.9　for 循环

只要测试表达式在循环体顶端计算时返回true，for循环结构就会执行循环体。for循环的语法如下所示，图9-8阐明了它。

- 在循环的开始，*Initializer*被执行一次。
- 然后*TestExpr*被求值。

- 如果它返回true，*Statement*被执行，接着是*IterationExpr*。
- 控制然后回到循环的顶端，*TestExpr*再次被求值。
- 只要*TestExpr*返回true，*Statement*和*IterationExpr*都将被执行。

```
               用分号隔开
              ↓          ↓
for( Initializer ; TestExpr ; IterationExpr )
    Statement
```

语句中的一些部分是可选的。

- *Initializer*、*TestExpr*和*IterationExpr*都是可选的。它们的位置可以空着。如果*TestExpr*位置是空的，那么测试被假定返回true。因此，如果程序要避免进入无限循环，必须有某种其他退出该语句的方法。
- 分号是必须的。

图9-8阐明了for语句的控制流。关于它的组成，还应了解下面这些内容：

- *Initializer*只被执行一次，在*for*结构的任何其他部分之前。它常被用于声明和初始化循环中使用的本地变量。
- *TestExpr*被求值以决定*Statement*是否应该被执行或跳过。它必须计算成bool型值。

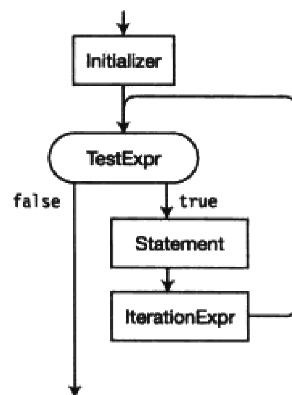



图9-8 for循环

- *IterationExpr*在*Statement*之后并且在返回到循环顶端到*Testexpr*之前立即被执行。

例如，在下面的代码中：

- 在任何其他事情之前，初始语句（int i=0）定义一个名称为i的变量，并初始化它的值为0。
- 然后条件（i<3）被求值。如果它为true，那么循环体被执行。
- 在循环的底部，在所有循环语句都被执行之后，*IterationExpr*语句被执行，本例中它递增i的值。

```
// The body of this for loop is executed three times.
for( int i=0 ; i<3 ; i++ )
   Console.WriteLine("Inside loop.  i:  {0}", i);

Console.WriteLine("Out of Loop");
```

这段代码产生以下输出：

```
Inside loop.  i:  0
Inside loop.  i:  1
Inside loop.  i:  2
Out of Loop
```

### 9.9.1 for语句中变量的有效范围

任何声明在*initializer*中的变量只在该for语句的内部可见。

- 这与C和C++不同，C和C++中声明把变量引入到外围的块。
- 下面的代码阐明了这点：

```
   这里需要类型来声明
        ↓
for( int i=0; i<10; i++ ) // 变量i在有效范围内
   Statement;             // 语句
                          // 在该语句之后，i不再存在

   这里仍需要类型，因为前面的变量i已经超出存在范围
        ↓
for( int i=0; i<10; i++ ) // 我们需要定义一个新的i变量
   Statement;             // 因为先前的i变量已经不存在
```

在循环体内部声明的变量只在循环体内部被认识。

**注意**　与C和C++不同，在初始语句中声明的变量的范围只维持到循环的长度。

### 9.9.2 初始化语句和迭代表达式中的多表达式

初始语句和迭代表达式都可以包含多个表达式，只要它们用逗号隔开。

例如，下面的代码在初始语句中有两个变量声明，而且在迭代表达式中有两个表达式：

```
static void Main( )
{
   const int MaxI = 5;
                  两个声明                    两个表达式
                     ↓                            ↓
   for (int i = 0, j = 10; i < MaxI; i++, j += 10)
   {
      Console.WriteLine("{0}, {1}", i, j);
   }
}
```

这段代码产生以下输出：

```
0, 10
1, 20
2, 30
3, 40
4, 50
```

## 9.10 跳转语句

当控制流到达跳转语句时，程序执行被无条件转移到程序的另一部分。跳转语句有：

- break
- continue
- return
- goto
- throw

这一章阐述前四条语句，throw语句在第11章讨论。

### 9.10.1 break 语句

在本章的前面部分你已经看到break语句被用在switch语句中。它还能被用在下列语句类型中：

- for
- foreach
- while
- do

在这些语句的语句体中，break导致执行跳出最内层封装的语句（innermost enclosing statement）。

例如，下面的while循环如果只靠它的测试表达式，将会是一个无限循环，它的测试表达式始终为true。但相反，在三次循环迭代之后，遇到了break语句，循环退出了。

```
int x = 0;
while( true )
{
    x++;
    if( x >= 3 )
        break;
}
```

### 9.10.2 continue 语句

continue语句导致程序执行转到下列类型循环的最内层封装循环（innermost enclosing statement）的顶端：

- while
- do
- for
- foreach

例如，下面的for循环被执行了五次，在前三次迭代中，它遇到continue语句并直接回到循环的顶部，错过了在循环底部的WriteLine语句。执行只在后两次迭代时才到达WriteLine语句。

```
for( int x=0; x<5; x++ )              // 执行循环5次
{
   if( x < 3 )                        // 先执行3次
      continue;                       // 直接回到循环开始处

   // 当 x≥3时执行下面的语句
   Console.WriteLine("Value of x is {0}", x);
}
```

这段代码产生以下输出：

```
Value of x is 3
Value of x is 4
```

下面的代码展示了一个continue语句在while循环中的示例。这段代码产生与前面*for*循环的示例相同的输出。

```
int x = 0;
while( x < 5 )
{
   if( x < 3 )
   {
      x++;
      continue;                       // 回到循环开始处
   }

   // 当 x≥3时执行下面的语句
   Console.WriteLine("Value of x is {0}", x);
   x++;
}
```

## 9.11　标签语句

标签语句由一个标识符后面跟着一个冒号再跟着一条语句组成。它有下面的形式。

```
Identifier: Statement
```

标签语句的执行完全如同标签不存在一样，并仅执行*Statement*部分。

- 给语句增加一个标签允许控制从代码的另一部分转移到该语句。
- 标签语句只允许用在块内部。

### 9.11.1　标签

标签有它们自己的声明空间，所以标签语句中的标识符可以是任何有效的标识符，包括那些可能已经在重叠的范围内声明的标识符，比如本地变量或参数名。

例如，下面的代码展示了标签的有效使用，该标签和一个本地变量有相同的标识符。

```
{
    int xyz = 0;                              // 变量 xyz
      ...
    xyz: Console.WriteLine("No problem.");    // 标签 xyz
}
```

然而，也有限制。该标识符不能是：

- 在重叠范围内和另一个标签标识符相同。
- 关键字。

## 9.11.2 标签语句的范围

标签语句不能从它的声明所在的块的外部可见（或可访问）。标签语句的范围是：

- 它声明所在的块。
- 任何嵌套在该块内部的块。

例如，图9-9左边的代码包含几个嵌套块，它们的范围被标记出来。在程序的范围B中声明了两个标签语句：increment和end。

- 图右边的阴影部分展示了该标签语句有效的代码区域。
- 在范围B和所有嵌套块中的代码都能看到并访问标签语句。
- 从范围内部任何位置，代码都能跳出到标签语句。
- 从外部（本例中范围A）代码不能跳入标签语句的块。

```
static void Main( )
{ // Scope A

    { // Scope B

    increment:  x++;
        { // Scope C

            { // Scope D
              ...
            }
            { // Scope E
              ...
            }
            ...
        }
    end: Console.WriteLine("Exiting");
    }
}
```

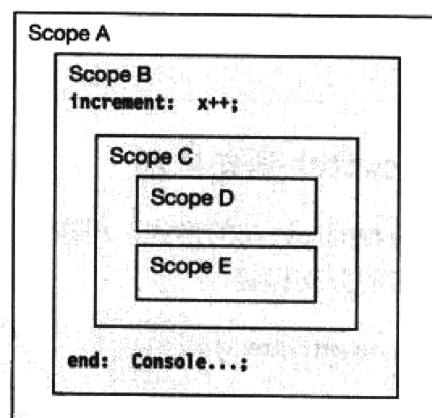



图9-9 标签的范围包括嵌套的块

# 9.12 goto 语句

goto语句无条件转移控制到一个标签语句。它的一般形式如下，其中Identifier是标签语句的标识符：

```
goto Identifier ;
```

例如，下面的代码展示了一个goto语句的简单使用：

```
bool thingsAreFine;
while (true)
{
   thingsAreFine = MonitorNuclearReactor();

   if ( thingsAreFine )
      Console.WriteLine("Things are fine.");
   else
      goto NotSoGood;
}

NotSoGood: Console.WriteLine("We have a problem.");
```

goto语句必须在标签语句的范围之内。

- goto语句可以跳到它本身所在块内的任何标签语句，或跳出到任何它被嵌套的块内的标签语句。
- goto语句不能跳入任何嵌套在该语句本身所在块内部的任何块。

**警告** 使用goto语句是非常不好的，因为它会导致弱结构化的、难以调试和维护的代码。Edsger Dijkstra在1968年给Communications of the ACM的信，标题为："Go To Statement Considered Harmful"，是对计算机科学非常重要的贡献。它是首先发表的描述使用goto语句的缺陷的文章之一。

### goto 语句在 switch 语句内部

还有另外两种goto语句的形式，用在switch语句内部。这些goto语句把控制转移到switch语句内部相应命名的分支标签。

```
goto case ConstantExpression;
goto default;
```

## 9.13 using 语句

某些类型的非托管对象有数量限制或很耗费系统资源。重要的是在代码使用完它们后，尽可能快地释放它们。using语句有助于简化该过程并确保这些资源被适当地处置。

资源是一个实现System.IDisposable接口的类或结构。接口在第17章详细阐述，但简而言之，接口就是未实现的函数成员的集合，类和结构可以选择去实现。IDisposable接口含有单独一个

名称为Dispose的方法。

使用资源的阶段如图9-10所示，它由以下部分组成：

- 分配资源。
- 使用资源。
- 处置资源。

如果在正在使用资源的那部分代码中产生一个意外的运行时错误，那么处置资源的代码可能得不到执行。

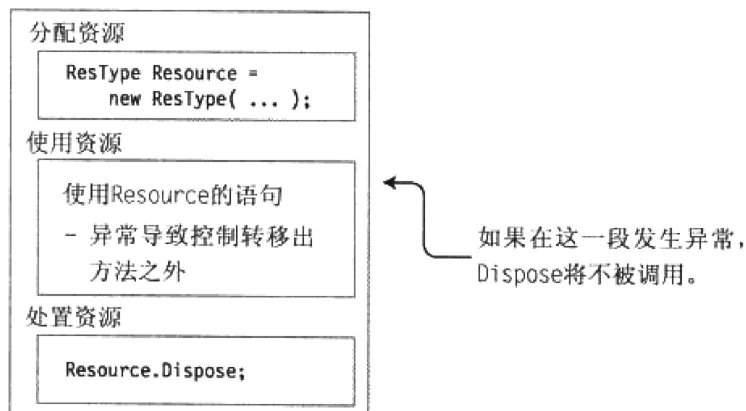



图9-10 使用资源的阶段

---

**说明** using语句不同于using指令。Using令词在第10章阐述。

---

## 9.13.1 资源的包装使用

using语句帮助减少意外的运行时错误带来的潜在问题，它整洁地包装了资源的使用。

有两种形式的using语句。第一种形式如下，图9-11阐明了它。

- 圆括号内的代码分配资源。
- Statement是使用资源的代码。
- using语句隐式产生处置该资源的代码。

```
using ( ResourceType Identifier = Expression ) Statement
                   ↑                               ↑
                分配资源                         使用资源
```

意外的运行时错误称为异常，异常在第11章阐述。处理可能的异常的标准方法是把可能导致异常的代码放进一个try块中，并把任何无论有没有异常都必须执行的代码放进一个finally块中。

这种形式的using语句确实是这么做的。它执行下列内容：

- 分配资源。

- 把Statement放进try块。
- 创建资源的Dispose方法的调用，并把它放进finally块。

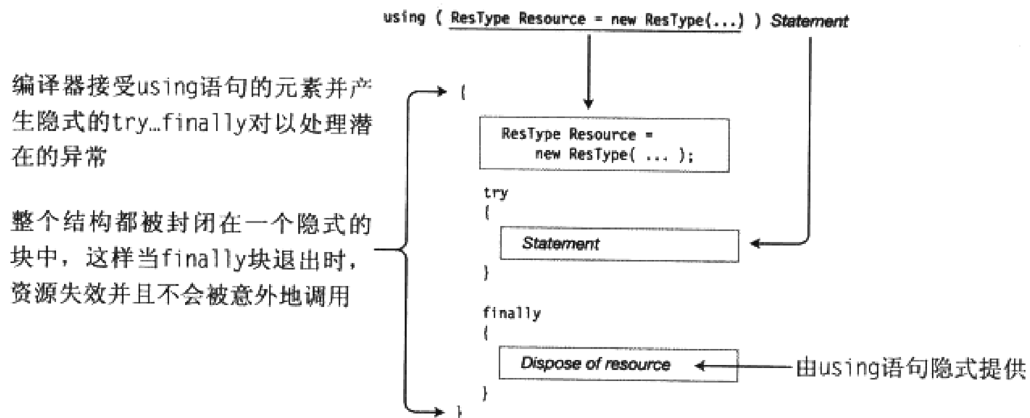



图9-11　using语句的效果

## 9.13.2　using 语句的示例

下面的代码使用using语句两次，一次对名称为TextWriter的类，一次对名称为TextReader的类，它们都来自System.IO命名空间。两个类都实现了IDisposable接口，这是using语句的要求。

- TextWriter资源打开一个文本文件来写，并向文件写一行。
- TextReader资源然后打开相同的文本文件，一行一行读取并显示它的内容。
- 在两种情况中，using语句确保对象的Dispose方法被调用。
- 还要注意到Main中的using语句和开始两行的using指令之间的区别。

```
using System;                    // using 指令，不是 using 语句
using System.IO;                 // using 指令，不是 using 语句

namespace UsingStatement
{
   class Program
   {
      static void Main( )
      {
         // using 语句
         using (TextWriter tw = File.CreateText("Lincoln.txt") )
         {
            tw.WriteLine("Four score and seven years ago, ...");
         }

         // using 语句
         using (TextReader tr = File.OpenText("Lincoln.txt"))
         {
            string InputString;
            while (null != (InputString = tr.ReadLine()))
```

```
                Console.WriteLine(InputString);
            }
        }
    }
}
```

这段代码产生以下输出:

```
Four score and seven years ago, ...
```

### 9.13.3 多个资源和嵌套

using语句还可以被用于相同类型的多个资源，资源声明用逗号隔开。语法如下:

```
            只有一个类型      资源            资源
                 ↓             ↓               ↓
using ( ResourceType Id1 = Expr1,  Id2 = Expr2, ... ) EmbeddedStatement
```

例如，在下面的代码中，每个using语句分配并使用两个资源。

```
static void Main()
{
   using (TextWriter tw1 = File.CreateText("Lincoln.txt"),
                     tw2 = File.CreateText("Franklin.txt"))
   {
      tw1.WriteLine("Four score and seven years ago, ...");
      tw2.WriteLine("Early to bed; Early to rise ...");
   }

   using (TextReader tr1 = File.OpenText("Lincoln.txt"),
                     tr2 = File.OpenText("Franklin.txt"))
   {
      string InputString;
      while (null != (InputString = tr1.ReadLine()))
         Console.WriteLine(InputString);
      while (null != (InputString = tr2.ReadLine()))
         Console.WriteLine(InputString);
   }
}
```

using语句还可以被嵌套。在下面的代码中，除了嵌套using语句以外，还要注意到没有必要对第二个using语句使用块，因为它仅由一条单独简单语句组成。

```
using ( TextWriter tw1 = File.CreateText("Lincoln.txt") )
{
   tw1.WriteLine("Four score and seven years ago, ...");

   using ( TextWriter tw2 = File.CreateText("Franklin.txt") ) // 嵌套语句
      tw2.WriteLine("Early to bed; Early to rise ...");       // 简单语句
}
```

### 9.13.4　using语句的另一种形式

using语句的另一种形式如下：

```
关键字    资源          使用资源
  ↓        ↓              ↓
using ( Expression ) EmbeddedStatement
```

在这种形式中，资源在using语句之前声明。

```
TextWriter tw = File.CreateText("Lincoln.txt");          // 声明资源

using ( tw )                                             // using 语句
   tw.WriteLine("Four score and seven years ago, ...");
```

虽然这种形式也能确保对资源的使用结束后Dispose方法总是被调用，但它不能防止你在using语句已经释放了它的非托管资源之后使用该资源，把它留在一种不一致的状态。因此它提供了较少的保护，而且不推荐使用。图9-12阐明了这种形式。

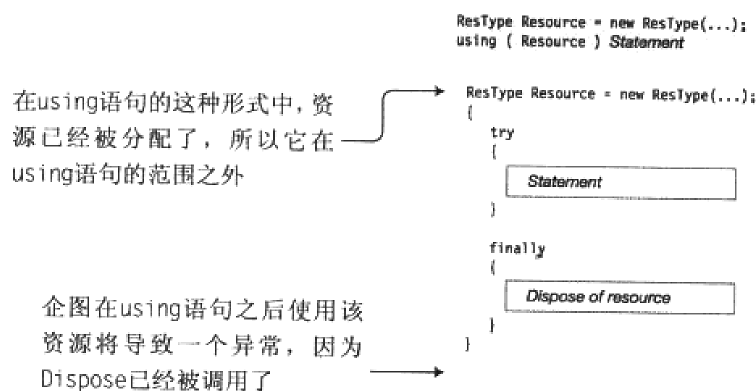



图9-12　资源声明在using语句之前

## 9.14　其他语句

有其他的语句和语言的特殊特征相关。这些语句在涉及那些特征的章节中阐述。在其他章中阐述的语句如表9-1所示。

**表9-1　在其他章中阐述的语句**

| 语　句 | 描　述 | 相关章节 |
|---|---|---|
| Checked，unchecked | 这些语句控制溢出检查上下文 | 第18章 |
| Foreach | 该语句遍历一个集合的每个成员 | 第14章和第20章 |
| try，throw，finally | 这些语句和异常有关 | 第11章 |
| return | 该语句返回控制到调用函数成员，而且还能返回一个值 | 第5章 |
| yield | 该语句用于迭代 | 第20章 |

# 第 10 章 命名空间和程序集

**本章内容**

- 引用其他程序集
- 命名空间
- using指令
- 程序集的结构
- 程序集标识符
- 强命名程序集
- 程序集的私有方式部署
- 共享程序集和GAC
- 配置文件
- 延迟签名

## 10.1 引用其他程序集

在第一章中，我们在高层次上观察了编译过程。编译器接受源代码文件并生称名称为程序集的输出文件。这一章中，我们将详细阐述程序集以及它们是如何生成和部署的。你还会看到命名空间是如何帮助组织类型的。

在迄今为止所看到的所有程序中，大部分都声明并使用它们自己的类。然而，在许多项目中，你会想使用来自其他程序集的类或类型。这些其他的程序集可能来自BCL，或来自一个第三方卖主，或你自己创建了它们。这些程序集称为类库，而且它们的程序集文件的名称通常以.dll扩展名结尾而不是.exe扩展名。

例如，假设你想创建一个类库，它包含可以被其他程序集使用的类和类型。一个简单库的源代码如下面示例中所示，它包含在名称为SuperLib.cs的文件中。该库含有单独一个名称为SquareWidget的公有类。图10-1阐明了DLL的生成。

```
public class SquareWidget
{
   public double SideLength = 0;
   public double Area
   {
```

```
            get { return SideLength * SideLength; }
        }
    }
```

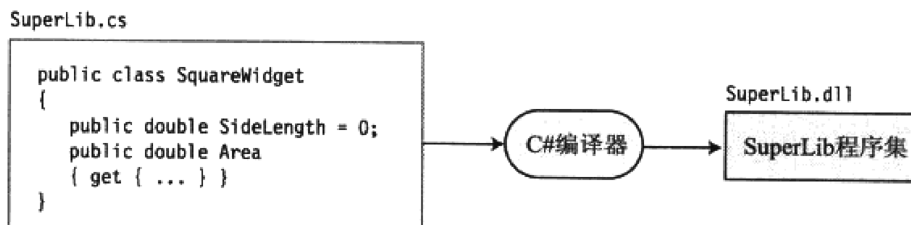



图10-1　SuperLib源代码和结果程序集

假设你还要写一个名称为MyWidgets的程序，而且你想使用SquareWidget类。程序的代码在一个名称为MyWidgets.cs的文件中，如下面的示例所示。这段代码简单创建一个类型SquareWidget的对象并使用该对象的成员。

```
using System;

class WidgetsProgram
{
    static void Main( )
    {
        SquareWidget sq = new SquareWidget();    // 来自类库
            ↑
        未在当前程序集中声明
        sq.SideLength = 5.0;                     // 设置边长
        Console.WriteLine(sq.Area);              // 输出该区域
    }      ↑
}       未在当前程序集中声明
```

注意，这段代码没有声明类SquareWidget。相反，使用的是定义在SuperLib中的类。然而，当你编译MyWidgets程序时，编译器必须知道你的代码在使用程序集SuperLib，这样它才能得到关于类SquareWidget的信息。要实现这点，需要给编译器一个到该程序集的引用，给出它的名称和位置。

在Visual Studio中，可以用下面的方法把引用添加到项目：

- 选择Solution Explorer并在该项目名下找到References目录。References目录包含项目使用的程序集的列表。
- 右键点击References目录并选择Add Reference。有五个可以从中选择的标签页，允许你以不同的方法找到类库。
- 对于我们的程序，选择Browse标签，浏览到包含SquareWidget类定义的DLL文件，并选择它。
- 点击OK按钮，引用就被加入到项目了。

在添加了引用之后，可以编译MyWidgets了。图10-2阐明了全部的编译过程。

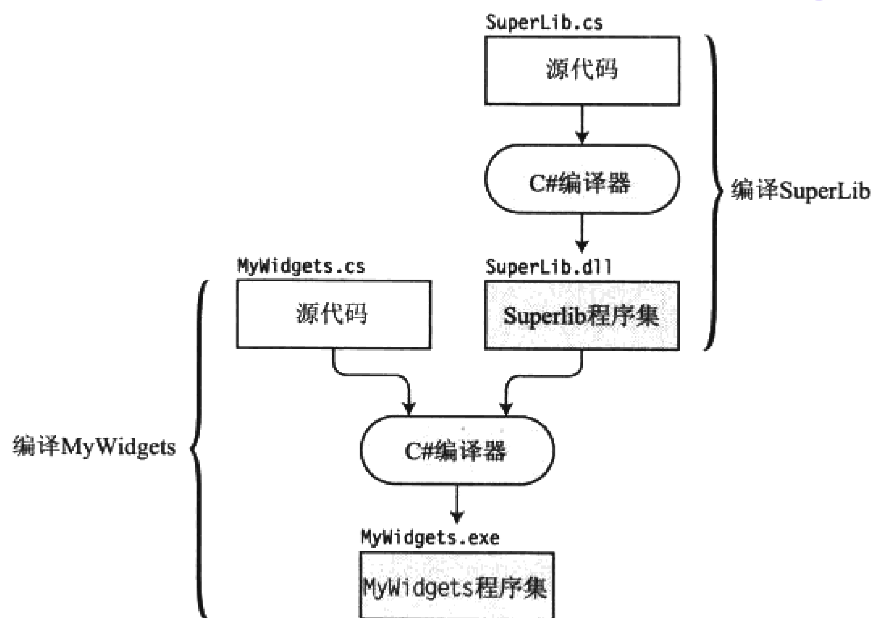



图10-2　引用另一个程序集

## mscorlib 库

有一个类库，我几乎在先前的每一个示例中都使用它。它就是包含Console类的那个库。Console类被定义在名称为mscorlib的程序集中，在名称为mscorlib.dll的文件里。然而，你不会看到这个程序集被列在References目录中。程序集mscorlib.dll含有C#类型以及大部分.NET语言的基本类型的定义。在编译C#程序时，它必须总是被引用，所以Visual Studio不把它显示在References目录中。

如果算上mscorlib，MyWidgets的编译过程看起来更像图10-3所示的表述。在此之后，我会假定使用mscorlib程序集而不再描述它。

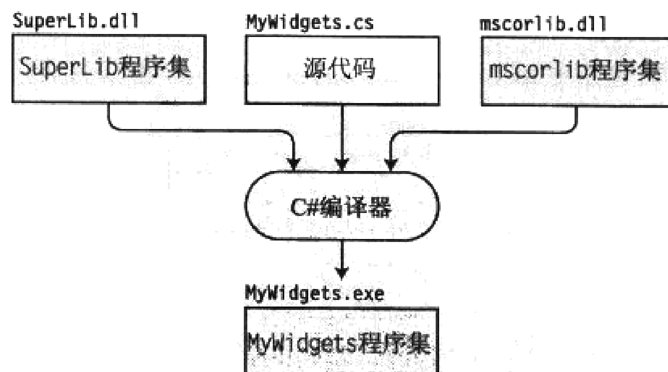



图10-3　引用类库

现在假设你的程序已经很好地用SquareWidget类工作了，但你想扩展它的能力以使用一个名称为CircleWidget的类，它被定义在另一个名称为UltraLib的程序集中。MyWidgets的源代码看上

去像下面这样。它创建一个SquareWidget对象和一个CircleWidget对象，它们分别定义在SuperLib中和UltraLib中。

```
class WidgetsProgram
{
   static void Main( )
   {
      SquareWidget sq = new SquareWidget();      // 来自 SuperLib
      ...

      CircleWidget circle = new CirclWidget();   // 来自 UltraLib
      ...
   }
}
```

类库UltralLib的源代码如下面的示例所示。注意，除了类CircleWidget之外，就像库SuperLib，它还声明了一个名称为SquareWidget的类。可以把UltraLib编译成一个DLL并加入到项目MyWidgets的引用列表中。

```
public class SquareWidget
{
   ...
}

public class CircleWidget
{
   public double Radius = 0;
   public double Area
   {
      get { ... }
   }
}
```

因为两个库都含有名称为SquareWidget的类，当你试图编译程序MyWidgets时，编译器产生一条错误消息，因为它不知道使用类SquareWidget的哪个版本。图10-4阐明了这种命名冲突。

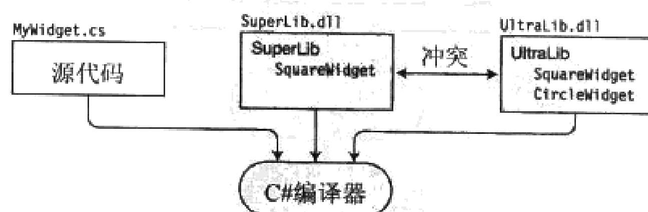



图10-4　由于程序集SuperLib和UltraLib都含有名称为SquareWidget的类的声明，编译器不知道实例化哪一个

## 10.2　命名空间

在MyWidgets示例中，由于你有源代码，你能通过在SuperLib源代码或UltraLib源代码中仅仅

改变SquareWidget类的名称来解决命名冲突。但是，如果这些类是由不同的公司开发的，而且你还不能拥有源代码会怎么样呢？假设SuperLib由一个名称为MyCorp的公司生产，UltraLib由ABCCorp公司生产。在这种情况下，如果你使用了任何有冲突的类或类型，你将不能把这两个库放在一起使用。

你能想象得出，在你做开发的机器上含有数打不同的公司生产的程序集，很可能有一定数量的类名重复。如果你不能把两个程序集用在一个程序中，仅仅因为它们碰巧有共同的类型名，这将是一种耻辱。命名空间特征须帮助你避免这个问题。

命名空间把一组类型分组在一起并给它们一个名称，称为*命名空间名称*。命名空间名称应该描述命名空间的内容并和其他命名空间名称相区别。

下面展示了声明一个命名空间的语法。声明在大括号中间的所有类和其他类型的名称都是命名空间的成员。

```
  关键字      命名空间名
    ↓             ↓
namespace SimpleNamespace
{
   TypeDeclarations
}
```

现在假设在MyCorp公司的程序员改变该源代码如下面的示例所示。它现在有一个环绕类声明的命名空间了。注意关于命名空间名称的两个有趣的事情：

- 命名空间可以包含前缀。
- 公司名称在命名空间名称的开始。

```
           公司名  句点
                 ↓  ↓
namespace MyCorp.SuperLib
{
   public class SquareWidget
   {
      public double SideLength = 0;
      public double Area
      {
         get { return SideLength * SideLength; }
      }
   }
}
```

当MyCorp公司运给你更新的程序集时，你可以通过修改你的MyWidgets程序如下面所示来使用它。注意，不仅使用类名（由于它在两个类库之间不明确）和使用命名空间名称做类名的前缀，并使用句点把它们隔开。带有命名空间名称和类名的整体字符串被称为*完全限定名*。

```
class WidgetsProgram
{
```

```
static void Main( )
{         完全限定名                                    完全限定名
    MyCorp.SuperLib.SquareWidget sq = new MyCorp.SuperLib.SquareWidget();
          命名空间名          类名

    CircleWidget circle = new CircleWidget();
    ...
```

现在你在代码中显式指定了SquareWidget的SuperLib版本，编译器不会再有区分类的问题了。完全限定名称输入起来有点长，但至少你现在能使用两个库了。在本章稍后，我会阐述using别名指令以解决不得不在完全限定名称中重复输入的麻烦。

如果UltraLib程序集也被生产它的公司（ABCCorp）使用命名空间更新，那么编译过程会如图10-5所示。

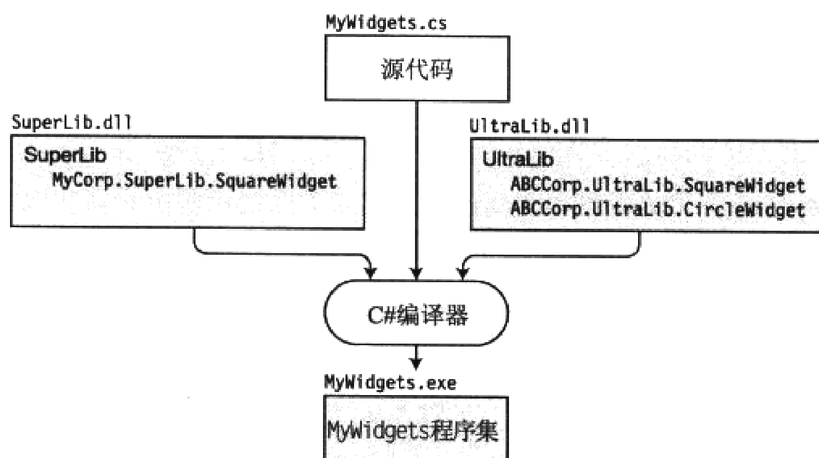



图10-5　带命名空间的类库

### 10.2.1　命名空间名称

如你所见，命名空间的名称可以包含创建该程序集的公司的名称。除了标识公司以外，该名称还用于帮助程序员获得定义在命名空间内的类型的种类的快速印象。

关于命名空间名称的一些要点如下：

- 命名空间名称可以是任何有效标识符。
- 命名空间名称可以包括句点符号，用于把类型组织成层次。

例如，表10-1列出了一些在.NET BCL中的命名空间的名称。

**表10-1　来自BCL的命名空间示例**

| | |
|---|---|
| System | System.IO |
| System.Data | Microsoft.CSharp |
| System.Drawing | Microsoft.VisualBasic |

下面是建议的命名空间命名指南：

- 使用公司名开始命名空间名称。
- 在公司名之后跟着技术名称。
- 不要把命名空间命名为与类或类型相同的名称。

例如，Acme Widget公司的软件开发部门在下面三个命名空间中开发软件，如下面的代码所示：

- AcmeWidgets.SuperWidget
- AcmeWidgets.Media
- AcmeWidgets.Games

```
namespace AcmeWidgets.SuperWidget.SPDComponent
{
   class SPDBase ...
   ...
}
```

### 10.2.2　命名空间的补充

关于命名空间，有其他几个要点应该知道：

- 在命名空间内，每个类型名必须有别于所有其他类型。
- 命名空间内的类型称为命名空间的成员。
- 一个源文件可以包含任意数目的命名空间声明，可以顺序也可以嵌套。

图10-6在左边展示了一个源文件，它顺序声明了两个命名空间，每个命名空间内有几个类型。注意，尽管命名空间内含有几个共有的类名，它们被它们的命名空间名称区分开来，如图右边的程序集所示。

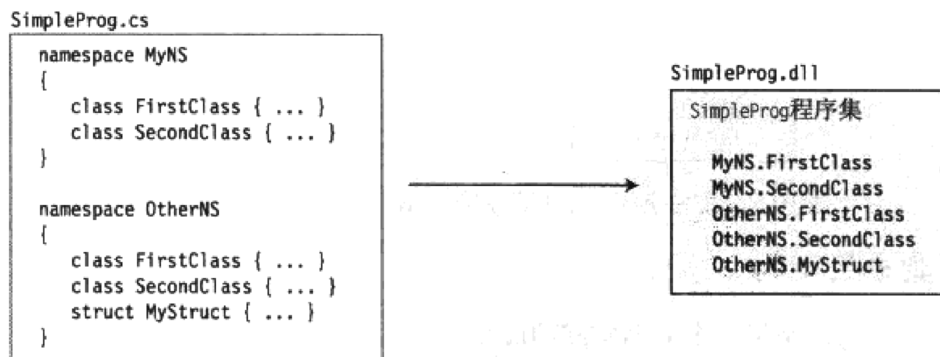



图10-6　多个命名空间在一个源文件中

.NET框架BCL提供了数千个已定义的类和类型以供在生成你的程序时选择。为了帮助组织这组有用的功能，相关功能的类型被声明在相同的命名空间里。BCL使用超过100个命名空间来组织它的类型。

### 10.2.3　命名空间跨文件伸展

命名空间不是封闭的。这意味着可以在该源文件的后面或另一个源文件中再次声明它，以对它增加更多的类型声明。

例如，图10-7展示了三个类的声明，它们都在相同的命名空间中，但声明在分离的源文件中。源文件可以被编译成单一的程序集，如图10-7所示，或编译成分离的程序集，如图10-8所示。

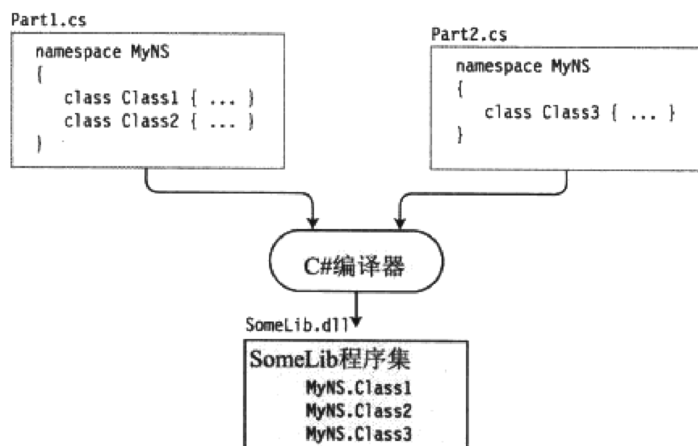



图10-7　命名空间可以跨源文件伸展并编译成单一程序集

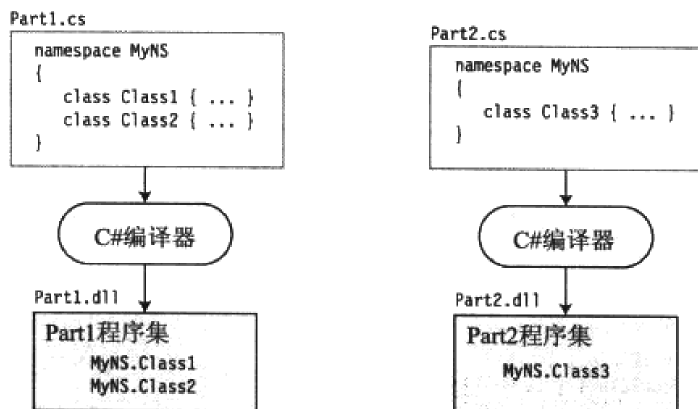



图10-8　命名空间可以跨源文件伸展并编译成分离的程序集

### 10.2.4　嵌套命名空间

一个命名空间可以是另一个命名空间的成员。这个成员被称为**嵌套的命名**空间。嵌套命名空间允许你创建类型的概念层次。

虽然嵌套的命名空间是外层命名空间的成员，但它的成员不是外层命名空间的成员。人们最初有对嵌套的命名空间的误解是既然嵌套的命名空间在外层命名空间内部，那么嵌套命名空间的成员必须是外层命名空间的子集。这是不对的，这些命名空间是分离的。

有两种方法声明一个嵌套的命名空间，如下所示。

- 原文嵌套：可以把命名空间的声明放在一个封装的命名空间声明体内部，从而创建一个嵌套的命名空间。图10-9的左边阐明了这种方法。在这个示例中，命名空间OtherNs嵌套在命名空间MyNamespace中。
- 分离的声明：也可以为嵌套命名空间创建分离的声明，但必须在声明中使用它的完全限定名称。图10-9的右边阐明了这种方法。注意在嵌套命名空间OtherNs的声明中，使用全路径命名MyNamespace.OtherNs。

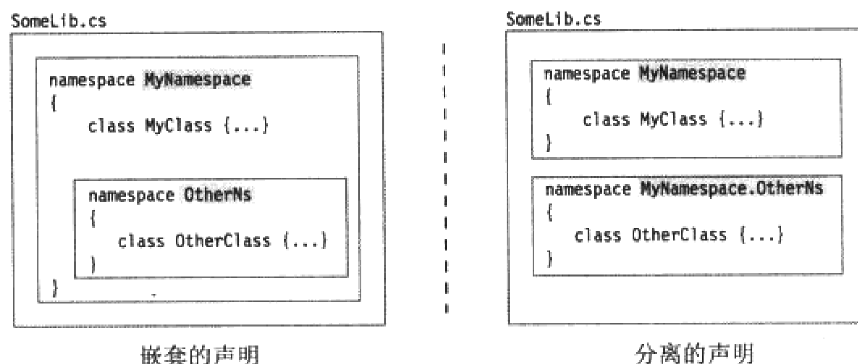



图10-9　声明嵌套命名空间的两种形式是等价的

图10-9所示的两种形式的嵌套命名空间声明生成相同的程序集，如图10-10所阐明的。该图展示了两个声明在SomeLib.cs文件中的类，使用它们的完全限定名。




图10-10　嵌套命名空间结构

## 10.3　using 指令

完全限定名可能相当长，在代码中通篇使用它们会变得十分乏味。然而，有两个编译指令，可以使你避免不得不使用完全限定名：using命名空间指令和using别名指令。

关于using指令词的两个要点如下：

- 它们必须放在源文件的顶端，在任何类型声明之前。
- 它们应用于当前源文件中的所有命名空间。

### 10.3.1　using 命名空间指令

在MyWidgets示例中，你看到多个部分使用完全限定名称指定一个类。可以通过在源文件的顶端放置using命名空间指令以避免不得不使用长名称。

using命名空间指令通知编译器你将要使用来自某个指定命名空间的类型。然后你可以继续，并使用简单类名而不必全路径修饰它们。

当编译器遇到一个不在当前命名空间的名称时，它检查在using命名空间指令中给出的命名空间列表，并把该未知名称加到列表中的第一个命名空间后面。如果结果完全限定名称匹配了这个程序集或引用程序集中的一个类，编译器将使用那个类。如果不匹配，那么它试验列表中下一个命名空间。

using命名空间指令由关键字using跟着一个命名空间标识符组成。

```
关键字
  ↓
using System ;
        ↑
  命名空间的名称
```

一个我已经在通篇文字中使用的方法是WriteLine方法，它是类Console的成员，在System命名空间中。不是在通篇代码中使用它的完全限定名，我只是简化了一点我们的工作，在代码的顶端使用using命名空间指令。

例如，下面的代码在第一行使用using命名空间指令以描述该代码使用来自System命名空间类或其他类型。

```
using System;                              // 命名空间指令
    ...
System.Console.WriteLine("This is text 1"); // 使用完全限定名称
Console.WriteLine("This is text 2");        // 使用指令
```

## 10.3.2　using别名指令

using别名指令允许起一个别名给：

- 命名空间
- 命名空间内的一个类型

例如，下面的代码展示了两个using别名指令的使用。第一个指令告诉编译器标识符Syst是命名空间System的别名。第二个指令表明标识符SC是类System.Console的别名。

```
关键字  别名  命名空间
  ↓      ↓      ↓
 using Syst = System;
 using SC   = System.Console;
  ↑      ↑      ↑
关键字  别名     类
```

下面的代码使用这些别名。在Main中三行代码都调用System.Console.WriteLine方法。

Main的第一条语句使用命名空间（System）的别名。

第二条语句使用该方法的完全限定名。

第三条语句使用类（Console）的别名。

```
using Syst = System;                       // using 别名指令
```

```
using SC    = System.Console;                    // using 别名指令

namespace MyNamespace
{
   class SomeClass
   {
      static void Main()
      {  命名空间的别名
           ↓
         Syst.Console.WriteLine  ("Using the namespace alias.");
         System.Console.WriteLine("Using fully qualified name.");
         SC.WriteLine            ("Using the type alias");
           ↑
      }  类的别名
   }
}
```

## 10.4 程序集的结构

如你在第1章看到的，程序集不包含本地机器代码，而是公共中间语言代码。它还包含实时编译器（JIT）在运行时转换CIL到本机代码所需的一切，包括对它所引用的其他程序集的引用。程序集的文件扩展名通常为.exe或.dll。

大部分程序集由一个单独的文件构成。图10-11阐明了程序集的四个主要部分。

- 程序集的清单包含：
  - 程序集名称标识符。
  - 组成程序集的文件列表。
  - 一个指示程序集中内容在哪里的地图。
  - 关于引用的其他程序集的信息。
- 类型元数据部分包含该程序集中定义的所有类型的信息。这些信息包含关于每个类型要知道的所有事情。
- CIL部分包含程序集的所有中间代码。
- 资源部分是可选的，但可以包含图形和语言资源。

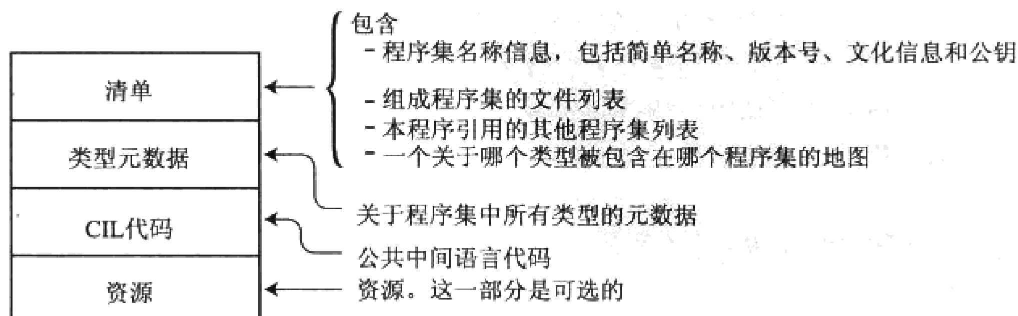



图10-11　单文件程序集的结构

尽管大部分程序集由单文件组成，但有些也有多个文件。对于有多个模块的程序集，一个文件是**主模块**（primary module），而其他的是**次要模块**（secondary modules）。

- 主模块含有程序集的清单和到次要模块的引用。
- 次要模块的文件名以扩展名.netmodule结尾。
- 多文件程序集被视为一个单一单元。它们一起部署并一起定版。

图10-12阐明了一个带次要模块的多文件程序集。

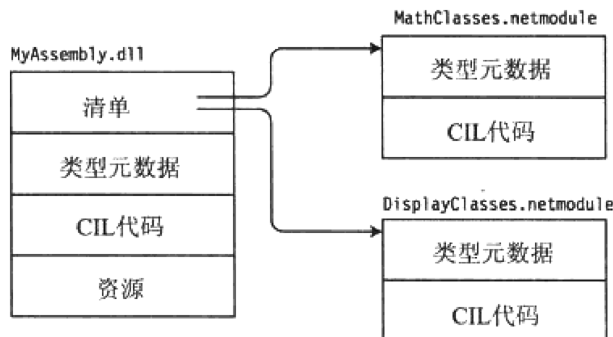



图10-12　多文件程序集

## 10.5　程序集标识符

在.NET框架中，程序集的文件名不像在其他操作系统和环境中那么重要，更重要的是程序集的标识符（identity）。

程序集的标识符有四个组成部分，它们一起唯一标识了该程序集，如下所示。

- **简单名**：这只是不带文件扩展名的文件名。每个程序集都有一个简单名。它也被称为程序集名或友好名称（friendly name）。
- **版本号**：它由四个句点分开的整数字符串组成，以MajorVersion.MinorVersion.Build.Revision的形式，例如2.0.35.9。
- **文化信息**：它是一个字符串，由2～5个字符组成，代表一种语言，或代表一种语言和一个国家或地区。例如，在美国使用英语的文化名是en-US。在德国使用德语，它是de-DE。
- **公钥**：这个128字节字符串应该是生产该程序集的公司唯一的。

公钥是公钥/私钥对的一部分，它们是一组两个非常大的、特别选择的数字，可以用于创建安全的数字签名。公钥，顾名思义，可以被公开。私钥必须被拥有者保护起来。公钥是程序集的标识符的一部分。我们稍后会在本章看到私钥的使用。

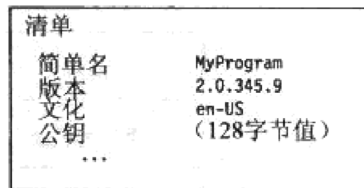



图10-13　清单中程序集标识符的组成部分

程序集名称的组成被包含在程序集清单中。图10-13阐明了清单部分。

图10-14展示了用在.NET文档和书籍中的关于程序集标识符的一些术语。

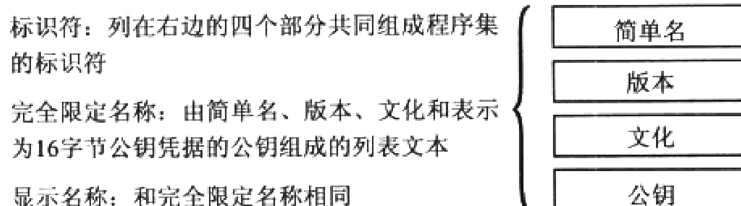



图10-14 关于程序集标识符的术语

## 10.6 强命名程序集

强命名（strongly named）程序集有一个唯一的数字签名依附于它。强命名程序集比没有强名称的程序集更加安全，这是由于以下原因：

- 强名称唯一标识了程序集。没有其他人能创建一个与之有相同名称的程序集，所以用户可以确信该程序集来自于其声称的来源。
- 没有CLR安全组件来捕获更改，带强名称的程序集的内容不能被改变。

弱命名（weakly named）程序集是没有被强命名的程序集。由于弱命名程序集没有数字签名，它天生是不安全的。因为一根链的强度只和它最弱的一环相同，所以强命名程序集默认只能访问其他强命名程序集（还存在一种方法允许"部分地相信调用者"，但我不会阐述这个主题）。

程序员不产生强名称。编译器产生它，接受关于程序集的信息，并哈希这些信息以创建一个唯一的数据签名依附到该程序集。它在哈希处理中使用的信息如下：

- 组成程序集的字节序列
- 简单名称
- 版本号
- 文化信息
- 公钥/私钥对

---

**说明** 在围绕强名称的命名法方面有一些差异。我所称作"强命名的"常指的是"强名称的"。我所称作"弱命名的"有时指的是"非强命名的"或"带简单名称的程序集"。

---

### 创建强命名程序集

要使用Visual Studio 2008强命名一个程序集，必须有一份公钥/私钥对文件的拷贝。如果没有密钥文件，可以让Visual Studio产生一个。可以实行以下步骤：

(1) 打开工程的属性。

(2) 选择签名页。

(3) 选择为程序集复选框签名并输入密钥文件的位置。

在编译代码时，编译器会生成一个强命名的程序集。编译器的输入和输出在图10-15中阐明。

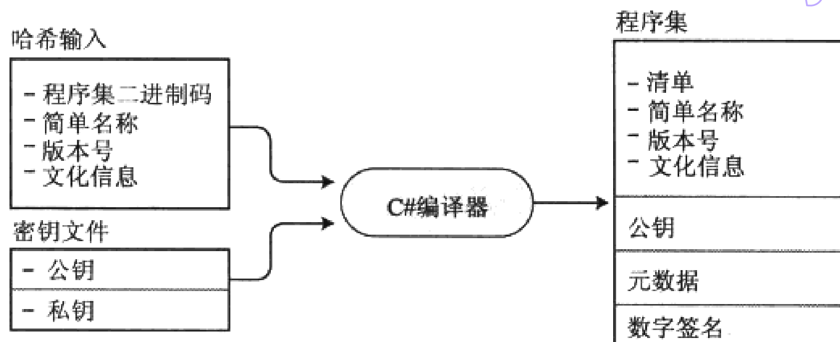



图10-15　创建强命名程序集

## 10.7　程序集的私有方式部署

在目标机器上部署一个程序就像在该机器上创建一个目录并把应用程序复制过去一样简单。如果应用程序不需要其他程序集（比如DLL），或如果所需的DLL在同一目录下，那么程序应该会就在它所在的地方良好工作。这种方法部署的程序集称为*私有程序集*，而且这种部署方法称为*复制文件*（XCopy）部署。

私有程序集几乎可以被放在任何目录中，而且只要它们依赖的文件都在同一目录或子目录下就足够了。事实上，可以在文件系统的不同部分有多个目录，每个目录都有同样的一组程序集，并且它们都会在它们各自不同的位置良好工作。

关于私有程序集部署的一些重要事情如下：

- 私有程序集所在的目录被称为*应用程序目录*。
- 私有程序集可以是强命名的也可以是弱命名的。
- 没有必要在注册表中注册组件。
- 要御载一个私有程序集，只要从文件系统中删除它即可。

## 10.8　共享程序集和 GAC

私有程序集是非常有用的，但有时你会想把一个DLL放在一个中心位置，这样一个单独的拷贝就能被系统中其他的程序集共享。.NET有这样的贮藏库，称为*全局程序集缓存*（GAC）。放进GAC的程序集称为*共享程序集*。

关于GAC的一些重要内容如下：

- 只有强命名程序集能被添加到GAC。
- 虽然GAC的早期版本只接受带`.dll`扩展名的文件，现在也可以添加带`.exe`扩展名的程序集了。
- GAC位置在名称为`Assembly`的子目录下，在Windows系统目录中。

### 10.8.1　把程序集安装到 GAC

当试图安装一个程序集到GAC时，CLR的安全组件首先必须检验程序集上的数字签名是否有

效。如果没有数据签名，或它是无效的，系统将不会把它安装到GAC。

然而，这是个一次性检查。在程序集已经在GAC内之后，当它被一个正在运行的程序引用时，不再需要进一步的检查。

gacutil.exe命令行工具允许从GAC添加或删除程序集，并列出GAC包含的程序集。它的三个最有用的参数标记如下所示。

- /i：把一个程序集插入GAC。
- /u：从GAC御载一个程序集。
- /l：列出GAC中的程序集。

## 10.8.2 GAC 内的并肩执行

在程序集部署到GAC之后，它就能被系统中其他程序集使用了。然而，请记住程序集的标识符由完全限定名称的全部四个部分组成。所以，如果一个库的版本号改变了，或如果它有一个不同的公钥，这些区别指定了不同的程序集。

结果就是在GAC中可以有许多不同的程序集，它们有相同的文件名。虽然它们有相同的文件名，它们是不同的程序集而且在GAC中完美地共存。这使不同的应用程序在同一时间很容易使用不同版本的同一DLL，因为它们是带不同的标识符的不同程序集。这被称为并肩执行（side-by-side Execution）。

图10-16阐明了GAC中四个不同的DLL，它们都有相同的文件名MyLibary.dll。看这个图，可以看出前三个来自于同一公司，因为它们有相同的公钥，第四个来自不同的来源，因为它有一个不同的公钥。这些版本像下面这样不同：

- 英文V1.0.0.0版，来自A公司。
- 英文V2.0.0.0版，来自A公司。
- 德文V1.0.0.0版，来自A公司。
- 英文V2.0.0.0版，来自B公司。

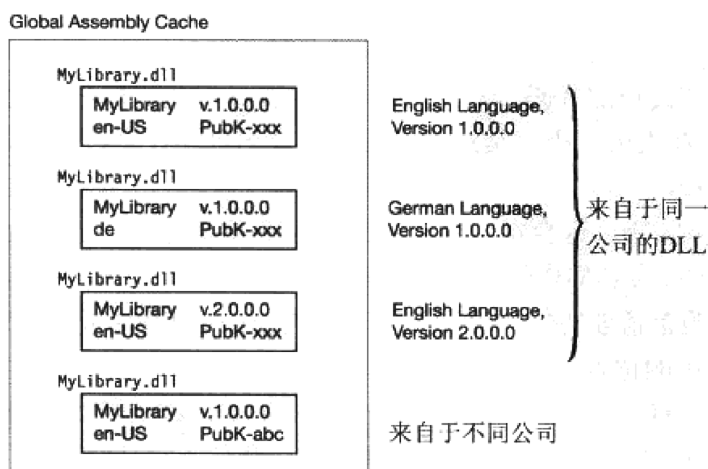



图10-16 在GAC中四个不同的并肩DLL

## 10.9　配置文件

配置文件含有关于应用程序的信息，供CLR在运行时使用。它们可以指示CLR去做这样的事情，比如使用一个不同版本的DLL，或搜索程序引用的DLL时在附加目录中查找。

配置文件由XML代码组成，并不包含C#代码。编写XML代码的细节超出了本书的范围，但应当理解配置文件的目的以及它们如何使用。它们的一种用途是更新一个应用程序集以使用新版本的DLL。

例如，假设有一个应用程序引用了GAC中的一个DLL。在应用程序的清单中，该引用的标识符必须完全匹配GAC中程序集的标识符。如果一个新版本的DLL发布了，它可以被添加到GAC中，在那里它可以幸福地和老版本共存。

然而，应用程序仍然在它的清单中包括老版本DLL的标识符。除非重新编译应用程序并使它引用新版本的DLL，否则它会继续使用老版本。如果这是你想要的，那也不错。

然而，如果你不想重新编译程序但又希望它使用新的DLL，那么你可以创建一个配置文件告诉CLR去使用新的版本而不是旧版本。配置文件被放在应用程序目录中。

图10-17阐明了运行时过程中的对象。左边的应用程序`MyProgram.exe`调用`MyLibrary.dll`的V1.0.0.0版，如点化线箭头所示。但应用程序有一个配置文件，而它指示CLR加载2.0.0.0版。注意配置文件的名称由可执行文件的全名（包括扩展名）加上附加扩展名`.config`组成。

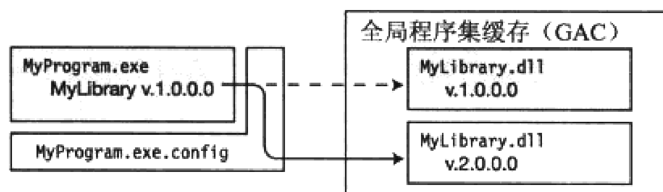



图10-17　使用配置文件绑定一个新版本

## 10.10　延迟签名

公司小心地保护它们官方的公钥/私钥对是非常重要的，否则，如果不可靠的人得到了它，就可以发布伪装成该公司的代码。为了避免这种情况，公司显然不能允许自由访问含有它们的公钥/私钥对的文件。在大公司中，最终程序集的强命名经常在开发过程的最尾部由特殊的有密钥访问权限的小组执行。

可是，由于个别原因，这会在开发和测试过程中导致问题。首先，由于公钥是程序集标识符的四个部分之一，所以直到提供了公钥它才能被设置。而且，弱命名的程序集不能被部署到GAC。开发人员和测试人员都需要有能力编译和测试该代码，并使用它将要被部署发布的方式，包括它的标识符和在GAC中的位置。

为了允许这个，有一种修改了的赋值强命名的形式，称为*延迟签名*（delayed signing）或*部分签名*（partial signing），它克服了这些问题，而且没有释放对私钥的访问。

在延迟签名中，编译器只使用公钥/私钥对中的公钥。然后公钥可以被放在完成的程序集的

标识符清单中。延迟签名还使用一个为0的块保留数字签名的位置。

要创建一个延迟签名的程序集，必须做两件事情。第一，创建一个密钥文件的拷贝，它只有公钥而不是公钥/私钥对。下一步，为程序集范围内的源代码添加一个名称为DelaySignAttribute的附加特性，并把它的值设为true。

图10-18展示了生成一个延迟签名程序集的输入和输出。注意图中下面的内容：

- 在输入中，DelaySignAttribute定位于源文件中，而且密钥文件只含有公钥。
- 在输出中，在程序集的底部有一个数字签名的保留空间。

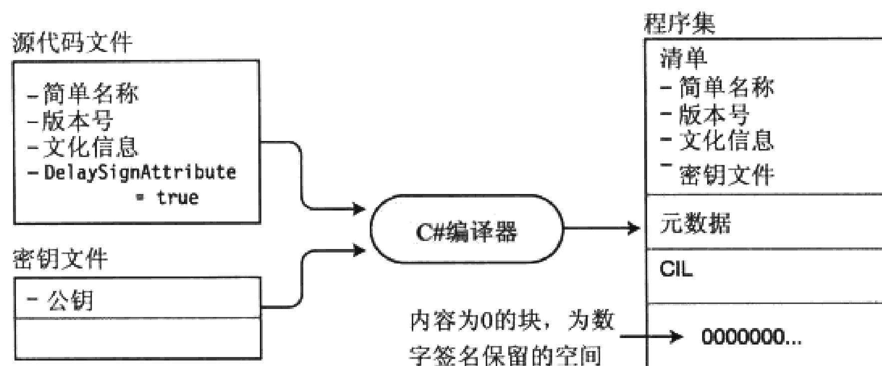



图10-18　创建延迟签名程序集

如果你试图部署延迟签名的程序集到GAC，CLR不会允许，因为它不是强命名的。要在这台机器上部署它，必须首先使用命令行指令取消在这台机器上的GAC签名确认，只针对这个程序集，并允许它被装在GAC中。要做到这点，从Visual Studio命令提示中执行下面的命令。

```
sn -vr MyAssembly.dll
```

现在，你已经看到弱命名程序集、延迟签名程序集和强签名程序集。图10-19总结了它们的结构区别。

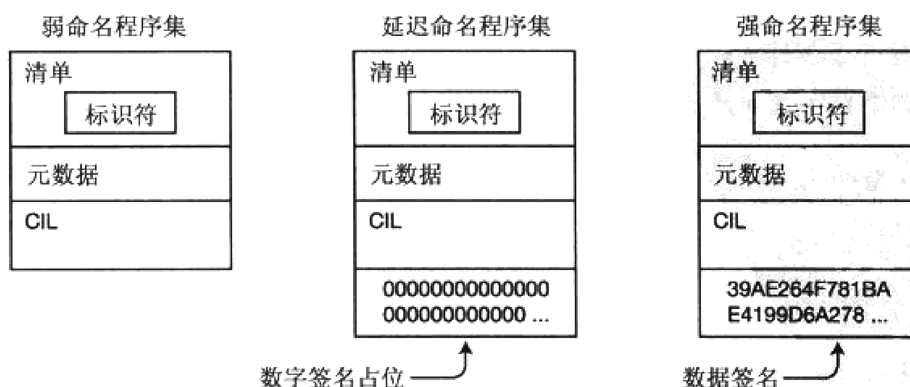



图10-19　不同程序集签名阶段的结构

# 第11章 异　　常

**本章内容**

- 什么是异常
- try语句
- 异常类
- catch子句
- finally块
- 为异常寻找处理代码
- 更进一步搜索
- 抛出异常
- 不带异常对象的抛出

## 11.1 什么是异常

异常是程序中的运行时错误，它违反了一个系统约束或应用程序约束，或出现了在正常操作时未预料的情形。例如，当程序试图用0除一个数或试图写一个只读文件时。当这些发生时，系统捕获这个错误并抛出（raise）一个异常。

如果程序没有提供处理该异常的代码，系统会挂起这个程序。

例如，下面的代码在试图用0除一个数时抛出一个异常：

```
static void Main()
{
   int x = 10, y = 0;
   x /= y;                // 用0除以一个数时抛出了一个异常
}
```

当这段代码运行时，系统显示下面的错误信息：

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.➥
      at Exceptions_1.Program.Main() in C:\Progs\Exceptions\Program.cs:line 12
```

## 11.2 try 语句

try语句用来指明被异常保护的代码块，并提供代码以处理异常，如果它们发生的话。try语句由三个部分组成，如图11-1所示。

- try块包含正被异常保护的代码。
- catch子句部分含有一个或多个catch子句。这些是处理异常的代码块。它们也称为是异常处理程序。
- finally块含有在所有情况下都要被执行的代码，无论有没有异常发生。

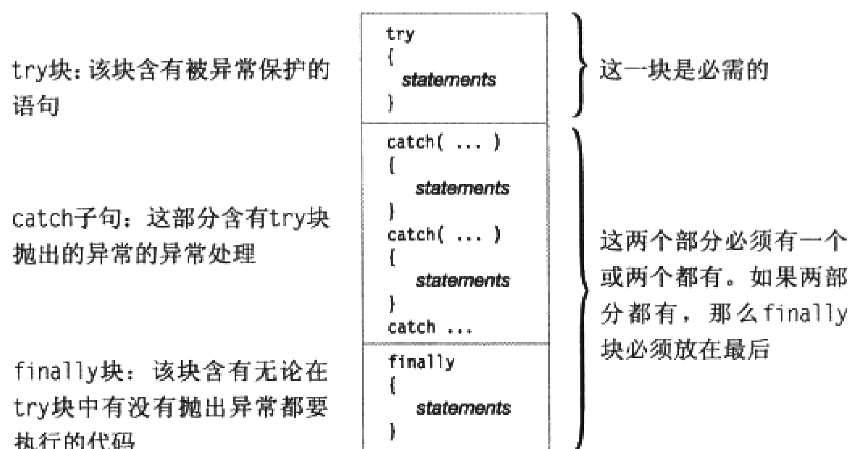



图11-1 try语句的结构

### 处理异常

前面的示例显示了除以0会导致一个异常。可以修改此程序，把那段代码放在一个try块中，并提供一个简单的catch子句，以处理该异常。当异常发生时，它被捕获并在catch块中处理。

```
static void Main()
{
   int x = 10;

   try
   {
      int y = 0;
      x /= y;                       // 抛出一个异常
   }
   catch
   {
      ...                           // 处理异常的代码

      Console.WriteLine("Handling all exceptions - Keep on Running");
   }
}
```

这段代码产生以下消息。注意，除了输出消息，没有异常已经发生的迹象。

```
Handling all exceptions - Keep on Running
```

## 11.3 异常类

有许多不同类型的异常可以在程序中发生。BCL定义了许多类，每一个类代表一个指定的异常类型。当一个异常发生时，CLR：

- 创建该类型的异常对象。
- 寻找适当的catch子句以处理它。

所有异常类都从根本上派生自System.Exception类。异常继承层次的一个部分如图11-2所示。

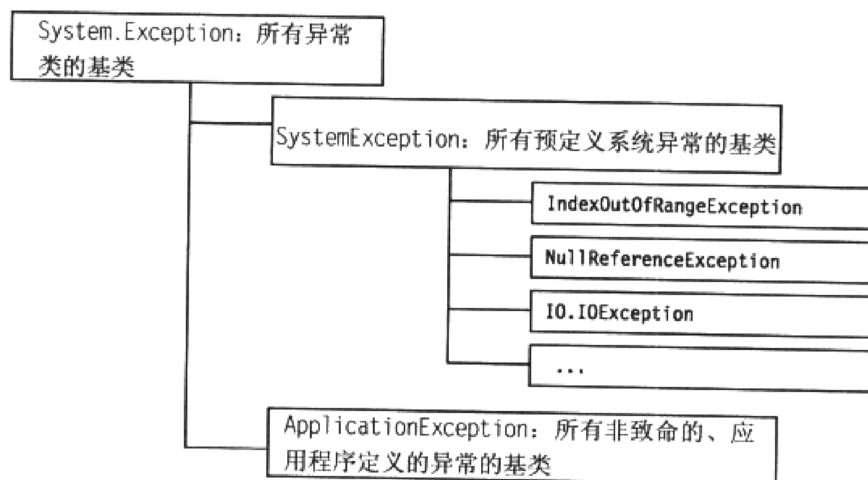



图11-2　异常层次的结构

异常对象含有只读属性，带有导致该异常的信息。这些属性的其中一些如表11-1所示。

**表11-1　挑选的异常对象属性**

| 属　性 | 类　型 | 描　述 |
|---|---|---|
| Message | string | 这个属性含有解释异常原因的消息 |
| StackTrace | string | 这个属性含有描述异常发生在何处的信息 |
| InnerException | Exception | 如果当前异常是由另一个异常引起的，这个属性包含前一个异常的引用 |
| HelpLink | string | 这个属性可以被应用程序定义的异常设置，为异常原因信息提供URN或URL |
| Source | string | 如果没有被应用程序定义的异常设定，那么这个属性含有异常起源所在的程序集的名称 |

## 11.4 catch子句

catch子句处理异常。它有三种形式，允许不同级别的处理。这些形式如图11-3所示。

```
catch                         一般catch子句
{                               - 在catch关键字之后没有参数列表
   Statements                   - 匹配try块中引起的任何类型的异常
}
------------------------------------------------------------------
                              特定catch子句
catch( ExceptionType )
{                               - 带有异常类的名称作为单一参数
   Statements                   - 匹配任何该名称类型的异常
}
------------------------------------------------------------------
                              带对象的特定catch子句
catch( ExceptionType InstID )   - 在异常类名称之后包括一个标识符
{                               - 该标识符在catch子句块中相当于一个本地变量，并被
   Statements                     称为异常变量
}                               - 异常变量引用异常对象，并能被用于访问关于该变量
                                  的信息
```

图11-3 catch子句的三种形式

一般catch子句能接受任何异常，但不能确定引发异常的类型。这只允许对任何可能发生的异常的普通处理和清理。

特定catch子句形式把一个异常类的名称作为参数。它匹配该指定类或派生自它的异常类的异常。

带对象的特定catch子句提供最多的关于异常的信息。它匹配该指定类的异常，或派生自它的异常类的异常。它还给出一个异常实例，称为**异常变量**，它是一个对CLR创建的异常对象的引用。可以在catch子句块内部访问异常变量的属性，以获取关于引起的异常的详细信息。

例如，下面的代码处理IndexOutOfRangeException类型的异常。当异常发生时，一个实际异常对象的引用被参数名e传入代码。那三个WriteLine语句中，每个都从异常对象中读取一个字符串字段。

```
                 异常类型          异常变量
                    ↓                 ↓
catch ( IndexOutOfRangeException   e )
{                                     访问异常变量
                                           ↓
   Console.WriteLine( "Message: {0}", e.Message );
   Console.WriteLine( "Source:  {0}", e.Source );
   Console.WriteLine( "Stack:   {0}", e.StackTrace );
```

### 11.4.1 使用特定 catch 子句的示例

回到除0的示例，下面的代码把前面的catch子句修改为指定处理DivideByZeroException类的异常。在前面的示例中，catch子句会处理所在try块中引起的任何异常，而这个示例将只处理DivideByZeroException类的异常。

```
int x = 10;
try
{
   int y = 0;
```

```
    x /= y;                   // 抛出一个异常
}           异常类型
               ↓
catch ( DivideByZeroException )
{
      ...
    Console.WriteLine("Handling an exception.");
}
```

可以进一步修改catch子句以使用一个异常变量。这允许在catch块内部访问异常对象。

```
int x = 10;
try
{
    int y = 0;
    x /= y;                   // 抛出一个异常
}           异常类型        异常变量
               ↓              ↓
catch ( DivideByZeroException  e )
{                                  访问异常变量
                                        ↓
    Console.WriteLine("Message: {0}", e.Message );
    Console.WriteLine("Source:  {0}", e.Source );
    Console.WriteLine("Stack:   {0}", e.StackTrace );
}
```

这段代码产生以下输出：

```
Message: Attempted to divide by zero.
Source:  Exceptions 1
Stack:      at Exceptions_1.Program.Main() in C:\Progs\Exceptions 1\➥
Exceptions 1\Program.cs:line 14
```

## 11.4.2　catch 子句段

catch子句段可以包含多个catch子句。图11-4展示了catch子句段的概貌。

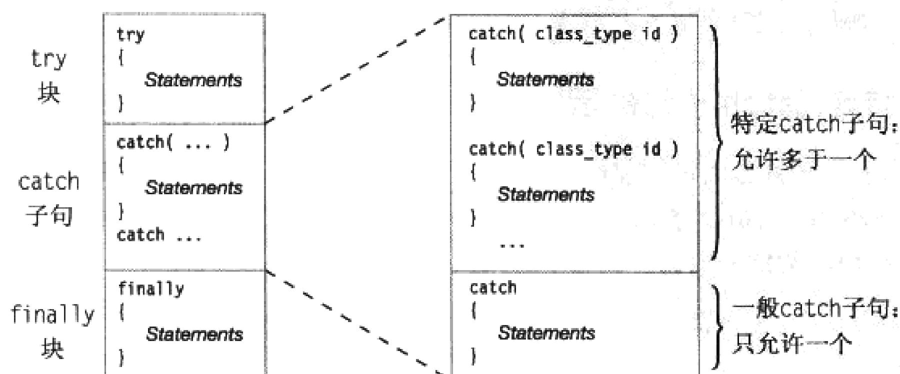



图11-4　try语句的catch子句段结构

当异常发生时，系统按顺序搜索catch子句的列表，第一个匹配该异常对象类型的catch子句被执行。因此，catch子句的排序有两个重要的规则。它们是：

- 特定catch子句必须以一种顺序排列，最明确的异常类型第一，直到最普通的类型。例如，如果声明了一个派生自NullRefrenceException的异常类，那么为派生异常类型的catch子句应该被列在NullReferenceException的catch子句之前。
- 如果有一个一般catch子句，它必须是最后一个，并且在所有特定catch子句之后。使用一般catch子句是不好的。应该尽可能使用特定catch子句。一般catch子句通过让程序继续执行隐藏错误，让程序处于一种未知的状态。

## 11.5 finally 块

如果程序的控制流进入了一个带finally块的try语句，那么finally始终会被执行。图11-5阐明了它的控制流。

- 如果在try块内部没有异常发生，那么在try块的结尾，控制流跳过任何catch子句并到finally块。
- 如果在try块内部发生了异常，那么在catch子句段中无论哪一个适当的catch子句被执行，接着就是finally块的执行。

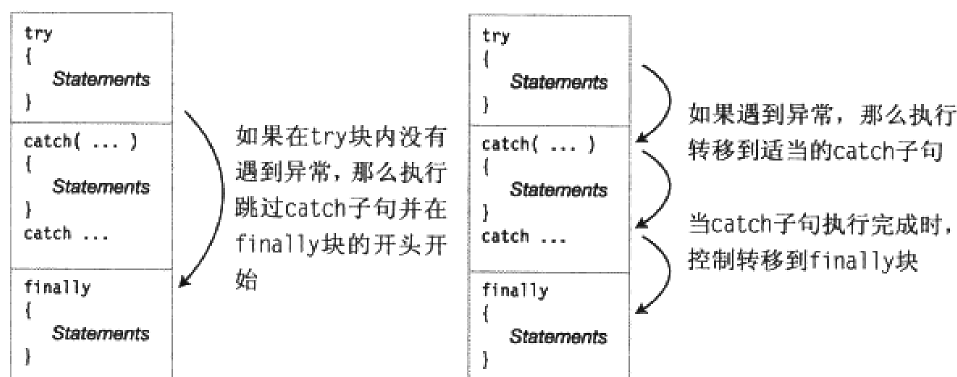



图11-5 finally块的执行

即使try块中有return语句，finally块也总是会在返回到调用代码之前执行。例如，在下面的代码中，在try块的中间有一条return语句，它在某条件下被执行。这不会使它绕过finally语句。

```
try
{
   if (inVal < 10) {
      Console.Write("First Branch  - ");
      return;
   }
   else
      Console.Write("Second Branch - ");
}
```

```
finally
{ Console.WriteLine("In finally statement"); }
```

这段代码在inVal值为5时产生以下输出：

```
First Branch  - In finally statement
```

## 11.6 为异常寻找处理代码

当程序产生一个异常时，系统查看该程序是否为它提供了一个处理代码。图11-6阐明了这个控制流。

- 如果在try块内发生了异常，系统会查看是否有任何一个catch子句能处理该异常。
- 如果找到了适当的catch子句：
  - 该catch子句被执行。
  - 如果有finally块，那么它被执行。
  - 执行在try语句的尾部继续（例如，在finally块之后，或如果没有finally块，就在最后一个catch子句之后）。

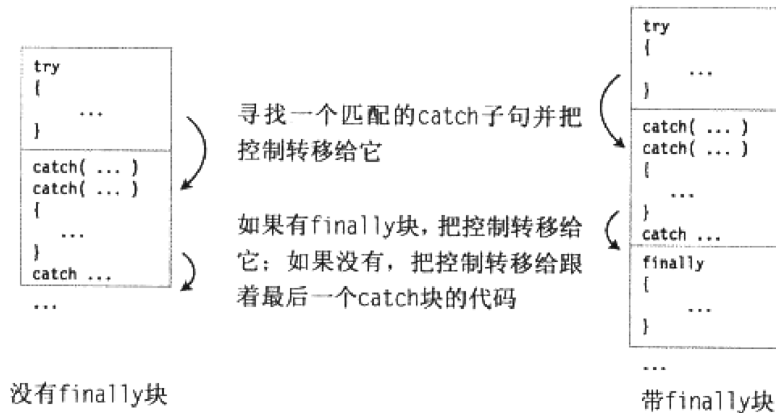



图11-6　在当前try语句中有处理代码的异常

## 11.7 更进一步搜索

如果异常在一个没有被try语句保护的代码段中产生，或如果try语句没有匹配的异常处理程序，系统将不得不更进一步寻找匹配的处理代码。为此它会按顺序搜索调用栈，以看看是否存在带匹配的处理代码的封装try块。

图11-7阐明了这个搜索过程。在图的左边是代码的调用结构，在右边是调用栈。该图显示Method2被从Method1的try块内部调用。如果异常发生在Method2内的try块内部，系统会：

- 首先，它查看Method2是否有能处理该异常的异常处理程序。
  - 如果有，Method2处理它，程序继续执行。
  - 如果没有，系统再延着调用栈找到method1，搜寻一个适当的处理代码。

- 如果Method1有一个适当的catch子句，那么系统将：
  - 回到栈顶，那里是Method2。
  - 执行Method2的finally块，并把Method2弹出栈。
  - 执行Method1的catch子句和它的finally块。
- 如果Method1没有适当的catch子句，系统会继续搜索调用栈。




图11-7 搜索调用栈

## 11.7.1 一般法则

图11-8展示了处理异常的一般法则。

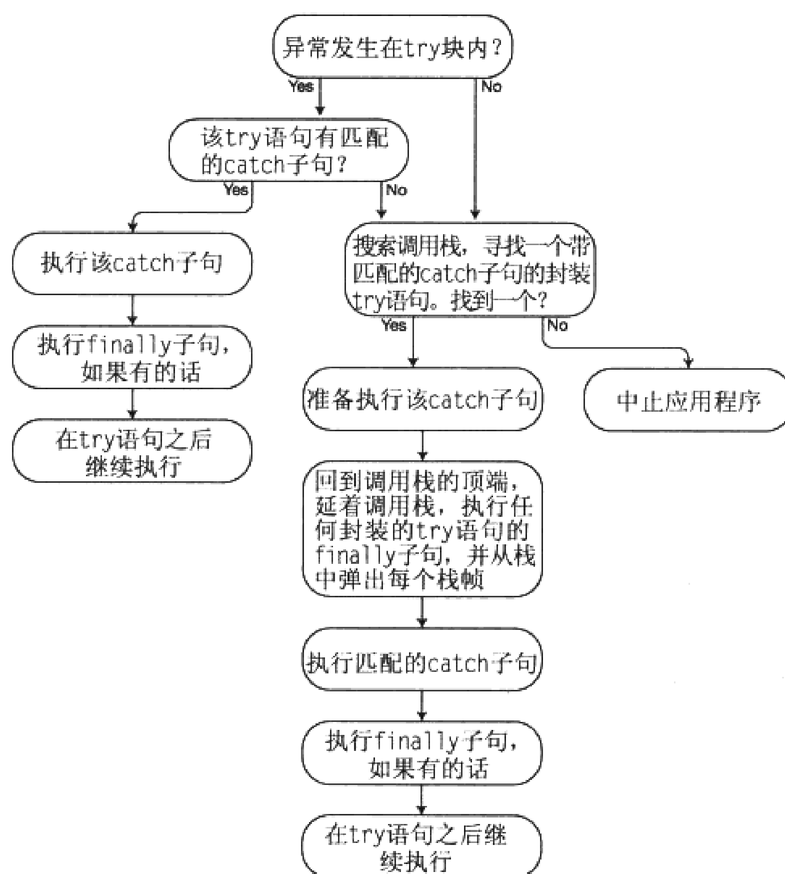



图11-8 处理异常的一般法则

## 11.7.2　搜索调用栈的示例

在下面的代码中，Main开始执行并调用方法A，A调用方法B。在代码之后，图11-9描述并图示了这个过程。

```
class Program
{
   static void Main()
   {
      MyClass MCls = new MyClass();
      try
         { MCls.A(); }
      catch (DivideByZeroException e)
         { Console.WriteLine("catch clause in Main()"); }
      finally
         { Console.WriteLine("finally clause in Main()"); }
      Console.WriteLine("After try statement in Main.");
      Console.WriteLine("            -- Keep running.");
    }
}

class MyClass
{
   public void A()
   {
      try
         { B(); }
      catch (System.NullReferenceException)
         { Console.WriteLine("catch clause in A()"); }
      finally
         { Console.WriteLine("finally clause in A()"); }
   }

   void B()
   {
      int x = 10, y = 0;
      try
         { x /= y; }
      catch (System.IndexOutOfRangeException)
         { Console.WriteLine("catch clause in B()"); }
      finally
         { Console.WriteLine("finally clause in B()"); }
   }
}
```

这段代码产生以下输出：

```
finally clause in B()
finally clause in A()
```

```
catch clause in Main()
finally clause in Main()
After try statement in Main.
          -- Keep running.
```

(1) Main调用A，A调用B，B遇到一个DivideByZeroExceprion异常。

(2) 系统检查B的catch段寻找匹配的catch子句。虽然它有一个IndexOutOfRangeException的子句，但没有DivideByZeroException的。

(3) 系统然后延着调用栈向下移动并检查A的catch段，在那里它发现A也没有匹配的catch子句。

(4) 系统继续延调用栈向下，并检查Main的catch子句部分，在那里它发现Main确实有一个DivideByZeroException的catch子句。

(5) 尽管匹配的catch子句现在被定位了，但并不执行。相反，系统回到栈的顶端，执行B的finally子句，并把B从调用栈中弹出。

(6) 系统移动到A，执行它的finally子句，并把A从调用栈中弹出。

(7) 最后，Main的匹配catch子句被执行，接着是它的finally子句。然后执行在Main的try语句结尾之后继续。

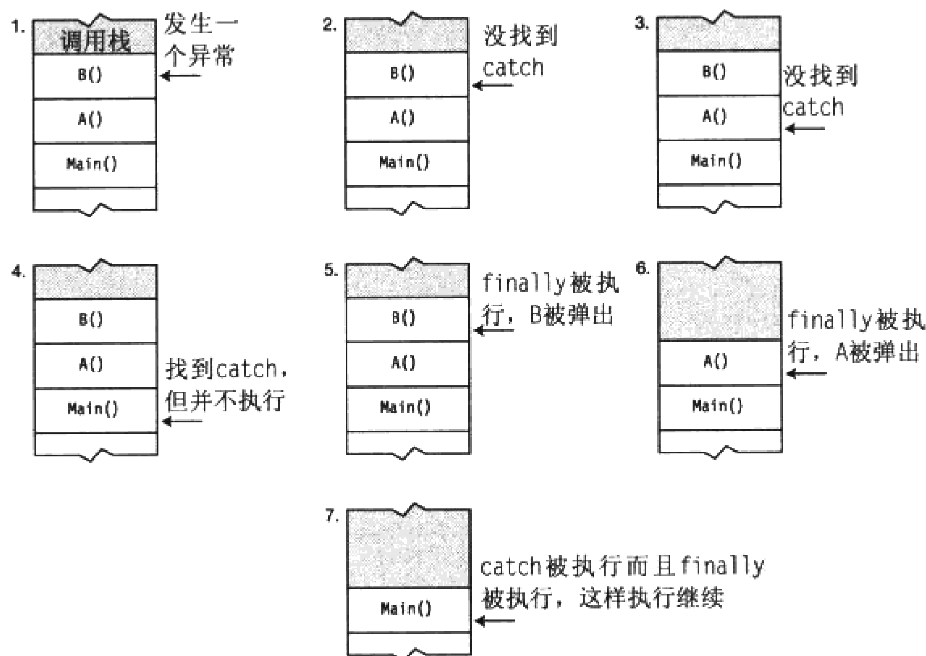



图11-9 搜索栈寻找一个异常处理程序

## 11.8 抛出异常

可以使用throw语句使代码显式地引发一个异常。throw语句的语法如下：

```
throw ExceptionObject;
```

例如，下面的代码定义了一个名称为PrintArg的方法，它带一个string参数并把它打印出来。在try块内部，它首先做检查以确认该参数不是null。如果它是null，它创建一个ArgumentNullException实例并抛出它。该异常实例在catch语句中被捕获，并且该出错消息被打印。Main调用该方法两次：一次用null参数，然后用一个有效参数。

```
class MyClass
{
   public static void PrintArg(string arg)
   {
      try
      {
         if (arg == null)                                  提供null参数的名称
         {                                                          ↓
            ArgumentNullException myEx = new ArgumentNullException("arg");
            throw myEx;
         }
         Console.WriteLine(arg);
      }
      catch (ArgumentNullException e)
      {
         Console.WriteLine("Message:  {0}", e.Message);
      }
   }
}
class Program
{
   static void Main()
   {
      string s = null;
      MyClass.PrintArg(s);
      MyClass.PrintArg("Hi there!");
   }
}
```

这段代码产生以下输出：

```
Message:  Value cannot be null.
Parameter name: arg
Hi there!
```

## 11.9　不带异常对象的抛出

throw语句还可以不带异常对象使用，在catch块内部。

- 这种形式重新抛出当前异常，系统继续它的搜索，为该异常寻找另外的处理代码。
- 这种形式只能用在catch语句内部。

例如，下面的代码从第一个catch子句内部重新抛出异常：

```
class MyClass
{
   public static void PrintArg(string arg)
   {
      try
      {
         try
         {
            if (arg == null)                                提供null参数的名称
            {                                                        ↓
               ArgumentNullException myEx = new ArgumentNullException("arg");
               throw myEx;
            }
            Console.WriteLine(arg);
         }
         catch (ArgumentNullException e)
         {
            Console.WriteLine("Inner Catch:  {0}", e.Message);
            throw;
         }   ↑
      }   重新抛出异常，没有附加参数
      catch
      {
         Console.WriteLine("Outer Catch:  Handling an Exception.");
      }
   }
}

class Program {
   static void Main() {
      string s = null;
      MyClass.PrintArg(s);
   }
}
```

这段代码产生以下输出：

```
Inner Catch:  Value cannot be null.
Parameter name: arg
Outer Catch:  Handling an Exception.
```

# 第12章 结　　构

**本章内容**

- ❑ 什么是结构
- ❑ 结构是值类型
- ❑ 对结构赋值
- ❑ 构造函数和析构函数
- ❑ 字段初始化是不允许的
- ❑ 结构是密封的
- ❑ 装箱和取消装箱
- ❑ 结构作为返回类型和参数
- ❑ 关于结构的附加信息

## 12.1 什么是结构

结构是程序员定义的数据类型，非常类似于类。它们有数据成员和函数成员。虽然与类相似，但是有许多重要的区别。最重要的区别是：

- ❑ 类是引用类型而结构是值类型。
- ❑ 结构是隐式密封的，这意味着它们不能被派生。

声明结构的语法与声明类相似。

```
 关键字
   ↓
struct StructName
{
   MemberDeclarations
}
```

例如，下面的代码声明了一个名称为Point的结构。它有两个公有字段，名称为X和Y。在Main中，结构类型Point有三个变量被声明，而且它们被赋值并打印出来。

```
struct Point
{
   public int X;
```

```
    public int Y;
}

class Program
{
    static void Main()
    {
        Point first, second, third;

        first.X  = 10; first.Y = 10;
        second.X = 20; second.Y = 20;
        third.X  = first.X + second.X;
        third.Y  = first.Y + second.Y;

        Console.WriteLine("first:  {0}, {1}", first.X,  first.Y);
        Console.WriteLine("second: {0}, {1}", second.X, second.Y);
        Console.WriteLine("third:  {0}, {1}", third.X,  third.Y);
    }
}
```

## 12.2 结构是值类型

和所有值类型一样，结构类型变量含有它自己的数据。从而：

- 结构类型的变量不能为null。
- 两个结构变量不能引用同一对象。

例如，下面的代码声明了一个名称为CSimple的类和一个名称为Simple的结构，并为它们各声明一个变量。图12-1展示了这两个变量将如何安排在内存中。

```
class CSimple
{
    public int X;
    public int Y;
}

struct Simple
{
    public int X;
    public int Y;
}

class Program
{
    static void Main()
    {
        CSimple cs = new CSimple();
        Simple  ss = new Simple();
           ...
```

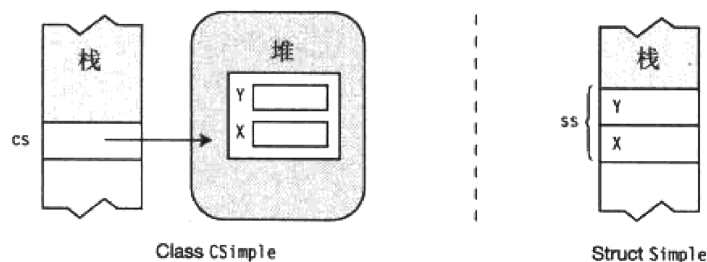



图12-1　类的内存排列对比结构的内存排列

## 12.3　对结构赋值

把一个结构赋值给另一个结构，就是从一个结构中把值复制到另一个结构。这和复制类变量不同，复制类变量时只有引用被复制。

图12-2展示了类变量赋值和结构变量赋值之间的区别。注意，在类赋值之后，cs2和cs1指向堆中同一对象。但在结构赋值之后，ss2的成员的值和ss1的相同。

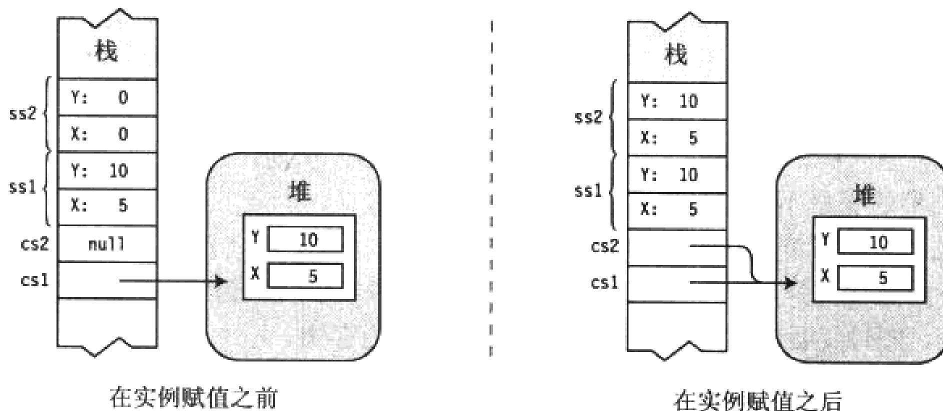



图12-2　类变量赋值和结构变量赋值

```
class CSimple
{ public int X; public int Y; }

struct Simple
{ public int X; public int Y; }

class Program
{
   static void Main()
   {
      CSimple cs1 = new CSimple(), cs2 = null;          // 类实例
      Simple  ss1 = new Simple(),  ss2 = new Simple();  // 结构实例

      cs1.X = ss1.X = 5;                     // 将5赋值给 ss1.X 和 cs1.X
      cs1.Y = ss1.Y = 10;                    // 将10赋值给 ss1.Y 和 cs1.Y
```

```
        cs2 = cs1;                       // 赋值类实例
        ss2 = ss1;                       // 赋值结构实例
    }
}
```

## 12.4 构造函数和析构函数

结构可以有实例构造函数和静态构造函数，但不允许有析构函数。

### 12.4.1 实例构造函数

语言隐式地为每个结构提供一个无参数的构造函数。这个构造函数把结构的每个成员设置为该类型的默认值。值成员被设置成它们的默认值，引用成员被设置成null。

预定义的无参数构造函数对每个结构都存在，而且不能删除或重定义它。但是，可以创建另外的构造函数，只要它们有参数。注意，这和类不同。对于类，编译器只在没有其他构造函数声明时提供隐式的无参数构造函数。

要调用一个构造函数，包括隐式无参数构造函数，要使用new运算符。注意，即使不从堆中分配内存也使用new运算符。

例如，下面的代码声明了一个简单的结构，它有一个带两个int参数的构造函数。Main创建两个该结构的实例，一个使用隐式无参数构造函数，第二个使用声明的两参数（two-parameter）构造函数。

```
struct Simple
{
   public int X;
   public int Y;

   public Simple(int a, int b)              // 带有参数的构造函数
   {
      X = a;
      Y = b;
   }
}

class Program
{
   static void Main()
   {                 调用隐式构造函数
                          ↓
      Simple s1 = new Simple();
      Simple s2 = new Simple(5, 10);
                          ↑
                     调用构造函数

      Console.WriteLine("{0},{1}", s1.X, s1.Y);
      Console.WriteLine("{0},{1}", s2.X, s2.Y);
   }
}
```

也可以不使用new运算符创建结构的实例。然而，如果这样做，有一些限制，它们如下：

- 不能使用数据成员的值，直到显式地设置它。
- 不能调用任何函数成员，直到所有数据成员已经被赋值。

例如，下面的代码展示了结构Simple的两个实例，它们没有使用new运算符创建。当企图访问s1而没有显式地设置该数据成员的值时，编译器产生一条错误消息。对s2的成员赋值之后，读取s2就没有问题了。

```
struct Simple
{
   public int X;
   public int Y;
}

class Program
{
   static void Main()
   {    没有构造函数的调用
              ↓  ↓
      Simple s1, s2;
      Console.WriteLine("{0},{1}", s1.X, s1.Y);           // 编译错误
                                    ↑     ↑
      s2.X = 5;                     还未被赋值
      s2.Y = 10;
      Console.WriteLine("{0},{1}", s2.X, s2.Y);           // OK
   }
}
```

### 12.4.2 静态构造函数

就像类，结构的静态构造函数创建并初始化静态数据成员，而且不能引用实例成员。结构的静态构造函数遵从与类的静态构造函数一样的规则。

静态构造函数在下面两个行为的第一个之前被调用：

- 调用显式声明的构造函数。
- 对结构静态成员的引用。

### 12.4.3 构造函数和析构函数的总结

表12-1总结了结构的构造函数和析构函数的使用。

表12-1　构造函数和析构函数的总结

| 类　型 | 描　述 |
|---|---|
| 实例构造函数（无参数） | 不能在程序中声明。系统为所有结构提供一个隐式的构造函数。它不能被程序删除或重定义 |
| 实例构造函数（有参数） | 可以在程序中声明 |
| 静态构造函数 | 可以在程序中声明 |
| 析构函数 | 不能在程序中声明。析构函数不被允许 |

## 12.5 字段初始化是不允许的

在结构中字段初始化是不允许的。

```
struct Simple
{              不允许
                 ↓
   public int x = 0;                          // 编译错误
   public int y = 10;                         // 编译错误
}                ↑
               不允许
```

## 12.6 结构是密封的

结构总是隐式密封的，因此，不能从它们派生其他结构。

由于结构不支持继承，个别类成员修饰符用在结构成员上将没有意义。因而，它们不能被用在结构成员声明中。不能用于结构的修饰符如下：

- protected
- internal
- abstract
- virtual

结构本身派生自System.ValueType，System.ValueType派生自object。

两个可以用于结构成员的继承相关的关键字是new和override修饰符，当创建一个和基类System.ValueType的成员有相同名称的成员时使用它们。所有结构都派生自System.ValueType。

## 12.7 装箱和取消装箱

如同其他值类型数据，如果想使用一个结构实例作为引用类型对象，必须执行装箱（Boxing）的复制。装箱和取消装箱（unboxing）在第18章详细阐述。

## 12.8 结构作为返回类型和参数

结构可以用作返回值和参数。

- 返回值：当结构被作为返回值时，一个拷贝被创建并从从函数成员返回。
- 值参：当结构被用作值参时，一个实际参数的拷贝被创建。该拷贝被用在方法的执行中。
- ref和out参数：如果把一个结构用作ref或out参数，一个对该结构的引用被传入方法，这样其数据成员就能被改变。

## 12.9 关于结构的附加信息

分配结构比创建类的实例需要更少的消耗，所以使用结构代替类有时可以提高性能，但要注

意到装箱和取消装箱的高代价。

最后，关于结构，需要知道的最后一些事情如下：

- 预定义简单类型（int、short、long，等等），尽管在.NET和C#中被视为原始类型，它们实际上在.NET中都被实现为结构。
- 可以使用声明partial类相同的方法声明partial结构，如第6章所述。
- 结构，就像类，可以实现接口。接口将在第17章阐述。

# 第 13 章 枚　举

**本章内容**

- 枚举
- 位标志
- 关于枚举的补充

## 13.1 枚举

枚举是由程序员定义的类型，与类或结构一样。

- 与结构一样，枚举是值类型，因此直接存储它们的数据，而不是分开存储成引用和数据。
- 枚举只有一种类型的成员：命名的整数值常量。

下面的代码展示了一个示例，声明了一个名称为TrafficLight的新枚举类型，它含有三个成员。注意成员声明列表是逗号分隔的列表，在枚举声明中没有分号。

```
关键字  枚举名称
  ↓        ↓
enum TrafficLight
{
    Green,    ← 逗号分隔，没有分号
    Yellow,   ← 转换成int
    Red
}
```

每个枚举类型都有一个底层整数类型，默认为int。

- 每个枚举成员被赋值一个底层类型的常量值。
- 编译器把第一个成员赋值为0，并对每一个后续成员赋的值比前一个成员多1。

例如，在TrafficLight类型中，编译器把int值0、1和2分别赋值给成员Green、Yellow和Red。在下面代码的输出中，可以看到底层的成员值，把它们转换成类型int。图13-1阐明了它们在栈中的排列。

```
TrafficLight t1 = TrafficLight.Green;
TrafficLight t2 = TrafficLight.Yellow;
TrafficLight t3 = TrafficLight.Red;
```

```
    Console.WriteLine("{0},\t{1}",   t1, (int) t1);
    Console.WriteLine("{0},\t{1}",   t2, (int) t2);
    Console.WriteLine("{0},\t{1}\n", t3, (int) t3);
                                          ↑
                                       转换成int
```

这段代码产生以下输出：

```
Green,  0
Yellow, 1
Red,    2
```

```
static void Main( )
{
   TrafficLight t1 = TrafficLight.Green;
   TrafficLight t2 = TrafficLight.Yellow;
   TrafficLight t3 = TrafficLight.Red;
}
```

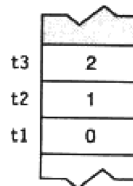



图13-1　枚举的成员常量被底层整数值表示

可以把枚举值赋给枚举类型变量。例如，下面的代码展示了三个TrafficLight类型变量的声明。注意可以把成员字面量赋给变量，或从另一个相同类型的变量复制值。

```
class Program
{
   static void Main()
   {     类型       变量          成员
          ↓          ↓             ↓
      TrafficLight t1 = TrafficLight.Red;         // 从成员赋值
      TrafficLight t2 = TrafficLight.Green;       // 从成员赋值
      TrafficLight t3 = t2;                       // 从变量赋值

      Console.WriteLine(t1);
      Console.WriteLine(t2);
      Console.WriteLine(t3);
   }
}
```

这段代码产生以下输出。注意，成员名被当作字符串打印。

```
Red
Green
Green
```

### 13.1.1　设置底层类型和显式值

可以通过把冒号和类型名放在枚举名之后以使用int以外的整数类型。类型可以是除了char以外的任何整数类型。所有成员常量都属于枚举的底层类型。

```
                    冒号
                     ↓
enum TrafficLight : ulong
{                       ↑
   ...               底层类型
```

成员常量的值可以是底层类型的任何值。要显式地设置一个成员的值，在枚举声明中的变量名之后使用初始化表达式。尽管不能有重复的名称，但可以有重复的值，如下所示。

```
enum TrafficLight
{
   Green  = 10,
   Yellow = 15,              // 重复的值
   Red    = 15               // 重复的值
}
```

例如，图13-2中的代码展示了两个枚举TrafficLight的等价的声明。

- 左边的代码接受默认的类型和编号。
- 右边的代码显式地设置底层类型为int，并设置成员为与默认值相应的值。

```
enum TrafficLight             |   enum TrafficLight : int
{                             |   {
   Green,                     |      Green  = 0,
   Yellow,                    |      Yellow = 1,
   Red                        |      Red    = 2
}                             |   }
```



图13-2 等价的枚举声明

### 13.1.2 隐式成员编号

可以显式地赋值给任何成员常量。如果不初始化一个成员常量，编译器隐式地给它赋一个值。图13-3阐明了编译器赋这些值使用的规则。

关联到成员名称的值不需要是独特的。

例如，下面的代码声明了两个枚举。CardSuit接受隐式的成员编号，如注释中所示。FaceCards显式地设置一些成员，而其他的接受隐式编号。

```
enum CardSuit
{
    Hearts,              // 0
    Clubs,               // 1
    Diamonds,            // 2
    Spades,              // 3
    MaxSuits             // 4
```

```
}

enum FaceCards
{
    // Member                // 所赋的值
    Jack            = 11,    // 11
    Queen,                   // 12
    King,                    // 13
    Ace,                     // 14
    NumberOfFaceCards = 4,   // 4
    SomeOtherValue,          // 5
    HighestFaceCard = Ace    // 14
}
```

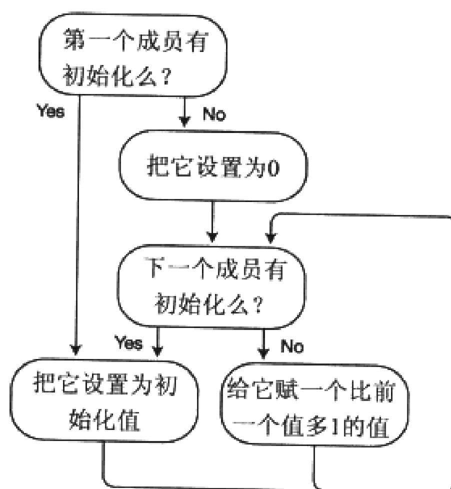



图13-3 成员赋值的法则

## 13.2 位标志

程序员们长期使用单字（single word）的不同位作为表示一组开/关标志的紧凑方法。枚举提供了实现它的简便方法。

一般的步骤如下：

(1) 确定需要多少个位标志，并选择一种有足够多位的无符号类型来保持它。

(2) 确定每个比特位置代表什么，并给它们一个名称。声明一个选中的整数类型的枚举，每个成员由一个比特位置表示。

(3) 使用按位或（OR）运算符设置保持该位标志的字中的适当的位。

(4) 使用按位与（AND）运算符解开位标志。

例如，下面的代码展示了枚举声明，表示纸牌游戏中一副牌的选项。底层类型uint足够满足四个位标志的需要了。注意代码的下列内容：

- 成员有表示二进制选项的名称。
    - 每个选项由字中的一个特殊的位表示。比特位置保持一个0或一个1。

- 因为一个位标志表示一个或开或关的比特，所以你不会想用0作为一个成员值。它已经有了一个意思：所有的位标志都是关。

❑ 用位模式时经常使用16进制表示法，因为与16进制表示法相比，位模式的10进制表示法有更直接的相关性。

❑ 使用Flags特性装饰（decorate）枚举实际上不是必要的，但可以有一些额外的便利，很快会讨论这一点。特性在第24章阐述。

```
[Flags]
enum CardDeckSettings : uint
{
   SingleDeck    = 0x01,          // 位 0
   LargePictures = 0x02,          // 位 1
   FancyNumbers  = 0x04,          // 位 2
   Animation     = 0x08           // 位 3
}
```

图13-4阐明了这个枚举。

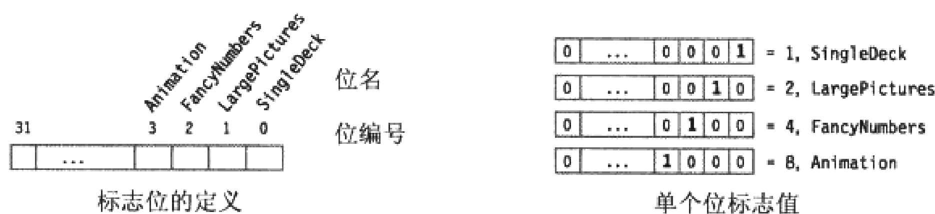



图13-4 标志位的定义和它们各自代表的值

要创建一个带有适当的位标志的字，需要声明一个该枚举类型的变量，并使用位或运算符设置需要的位。例如，下面的代码设置了四个选项中的三个：

```
   枚举类型        标志字           位标志被“或”在一起
      ↓               ↓                    ↓
CardDeckSettings ops = CardDeckSettings.SingleDeck
                     | CardDeckSettings.FancyNumbers
                     | CardDeckSettings.Animation ;
```

要确定是否有一个特定的位被设置了，对标志字和该位标志使用位与运算符。

例如，下面的代码检查一个值是否设置了FancyNumbers位标志。它通过把该值和位标志相与，然后比较结果和位标志来实现它。如果在原始值中该位被设置了，那么与操作的结果将和位标志有相同的位模式。

```
bool useFancyNumbers =
     (ops & CardDeckSettings.FancyNumbers) == CardDeckSettings.FancyNumbers;
       ↑                   ↑
     标志字               位标志
```

图13-5阐明了创建一个标志字然后检查一个特定位是否被设置的过程。

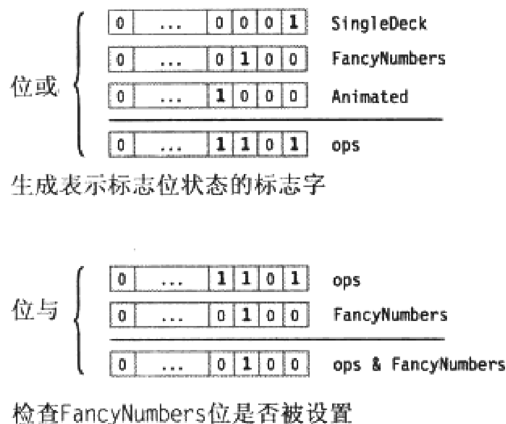



图13-5　生成一个标志字并检查一个特定的位

### 13.2.1　Flags 特性

我们将在第24章阐述特性，但这里应该提及Flags属性。特性看起来是放在类声明的上面一行的在方括号之间的字符串。特性根本不能改变计算。但是，它提供几个便利特征。

第一，它通知编译器、对象浏览器和其他观察这段代码的工具该枚举的成员将要被结合为位标志，而不是只作为分开的值使用。这使浏览器更合适地解释该枚举类型的变量。

第二，它允许一个枚举的ToString方法，以对位标志的值提供更合适的格式化。ToString方法接受一个枚举值并把它和枚举的常量成员的值做比较。如果它匹配了其中的一个成员，ToString返回该成员的字符串名称。

假设，例如使用CardDeckSettings（在前面的代码中给出）的枚举声明，并且不使用Flags特性。下面代码的第一行创建了一个该枚举类型的变量（名称为ops），并设置一个单一标志位的值。第二行使用ToString以获取该值代表的成员的字符串名称。

```
CardDeckSettings ops = CardDeckSettings.FancyNumbers;    // 设置位标志
Console.WriteLine( ops.ToString() );                      // 输出它的名称
```

这段代码产生以下输出：

```
FancyNumbers
```

一切都很好，但假设你设置了两个位标志而不是一个，如下面的代码所示：

```
// 设置两个位标志
ops = CardDeckSettings.FancyNumbers | CardDeckSettings.Animation;
Console.WriteLine( ops.ToString() );          // 输出什么？
```

ops的结果值是12，其中4来自于FancyNumbers标志，8来自于Animation标志。在第二行，当ToString试图在枚举成员列表中查找该值的时候，它发现没有值是12的成员，所以它只返回代表

12的字符串。结果输出如下：

```
12
```

然而，如果在枚举声明之前使用Flags特性，这告诉ToString方法位可以被分别对待。在查找该值时，它会发现12符合两个位标志成员FancyNumbers和Animation。然后它会返回包含它们的名称的字符串，用逗号和空格隔开，如下所示：

```
FancyNumbers, Animation
```

### 13.2.2 使用位标志的示例

下面的代码把所有使用位标志的内容放在一起：

```
[Flags]
enum CardDeckSettings : uint
{
    SingleDeck    = 0x01,          // 位 0
    LargePictures = 0x02,          // 位 1
    FancyNumbers  = 0x04,          // 位 2
    Animation     = 0x08           // 位 3
}

class MyClass
{
    bool UseSingleDeck = false;
    bool UseBigPics    = false;
    bool UseFancyNums  = false;
    bool UseAnimation  = false;

    public void SetOptions(CardDeckSettings ops)
    {
        UseSingleDeck = (ops & CardDeckSettings.SingleDeck)
                              == CardDeckSettings.SingleDeck;
        UseBigPics    = (ops & CardDeckSettings.LargePictures)
                              == CardDeckSettings.LargePictures;
        UseFancyNums  = (ops & CardDeckSettings.FancyNumbers)
                              == CardDeckSettings.FancyNumbers;
        UseAnimation  = (ops & CardDeckSettings.Animation)
                              == CardDeckSettings.Animation;
    }

    public void PrintOptions()
    {
        Console.WriteLine("Option settings:");
        Console.WriteLine("   Use Single Deck    - {0}", UseSingleDeck);
        Console.WriteLine("   Use Large Pictures - {0}", UseBigPics);
        Console.WriteLine("   Use Fancy Numbers  - {0}", UseFancyNums);
```

```
        Console.WriteLine("   Show Animation      - {0}", UseAnimation);
      }
   }

   class Program
   {
      static void Main() {
         MyClass mc = new MyClass();
         CardDeckSettings ops = CardDeckSettings.SingleDeck
                                | CardDeckSettings.FancyNumbers
                                | CardDeckSettings.Animation;
         mc.SetOptions(ops);
         mc.PrintOptions();
      }
   }
```

这段代码产生以下输出：

```
Option settings:
   Use Single Deck     - True
   Use Large Pictures  - False
   Use Fancy Numbers   - True
   Show Animation      - True
```

## 13.3　关于枚举的补充

枚举只有单一的成员类型：声明的成员常量。

- 不能对成员使用修饰符。它们都隐式地具有和枚举相同的可访问性。
- 由于成员是常量，即使在没有该枚举类型的变量时它们也可以访问。使用枚举类型名，跟着一个点和成员名。

例如，下面的代码没有创建枚举TrafficLight类型的任何变量，但它的成员是可访问的，并且可以使用WriteLine打印。

```
static void Main()
{
   Console.WriteLine("{0}", TrafficLight.Green);
   Console.WriteLine("{0}", TrafficLight.Yellow);
   Console.WriteLine("{0}", TrafficLight.Red);
}                              ↑          ↑
                            枚举名称   成员名称
```

枚举是一个独特的类型。比较不同枚举类型的成员会导致一个编译期错误。例如，下面的代码声明了两个枚举类型。

第一个if语句是正确的，因为它比较同一枚举类型的不同成员。

第二个if语句产生一个错误，因为它比较来自于不同枚举类型的成员，尽管它们的结构和成员名称完全相同。



```
enum FirstEnum                             // 第一个枚举类型
{
   Mem1,
   Mem2
}

enum SecondEnum                            // 第二个枚举类型
{
   Mem1,
   Mem2
}

class Program
{
   static void Main()
   {
      if (FirstEnum.Mem1 < FirstEnum.Mem2)  // 正确，相同的枚举类型
         Console.WriteLine("True");

      if (FirstEnum.Mem1 < SecondEnum.Mem1) // 错误，不同的枚举类型
         Console.WriteLine("True");
   }
}
```

# 第 14 章 数　组

14

## 本章内容

- 数组
- 数组的类型
- 数组是对象
- 一维数组和矩形数组
- 实例化一维数组或矩形数组
- 访问数组元素
- 初始化数组
- 交错数组
- 比较矩形数组和交错数组
- foreach语句
- 数组协变
- 数组继承的有用成员
- 比较数组类型

## 14.1　数组

数组实际上是由一个变量名称表示的一组同类型的数据元素。每个元素通过变量名称和一个或多个方括号中的索引名称来访问，如下所示：

```
数组名  索引
  ↓     ↓
MyArray[4]
```

### 14.1.1　定义

让我们从C#中与数组有关的一些重要定义开始。

- 元素：数组的独立数据项被称作元素。数组的所有元素必须是相同类型的，或继承自相同的类型。

- 秩/维度：数组可以有任何为正数的维度数。数组的维度数称作秩（rank）。
- 维度长度：数组的每一个维度有一个长度，就是这个方向的位置个数。
- 数组长度：数组的所有维度中的元素的总和称为数组的长度。

## 14.1.2　重要细节

下面是有关C#数组的一些要点：

- 数组一旦被创建，大小就固定了。C#不支持动态数组。
- 数组索引号是从0开始的。也就是说，如果维度长度是$n$，索引号范围是从0到$n-1$。例如，图14-1演示了两个示例数组的维度和长度。注意，对于每一个维度，索引范围从0到长度–1。

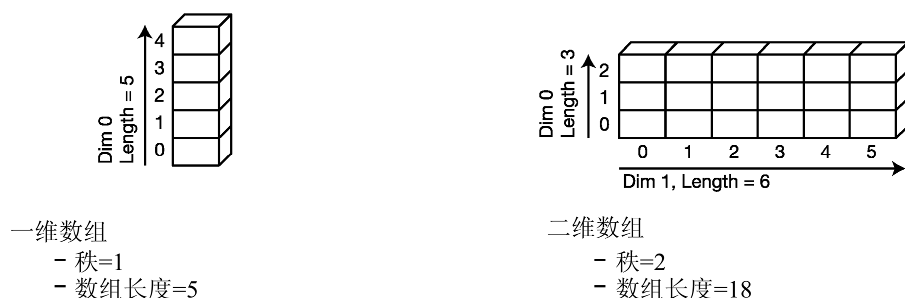



图14-1　维度和大小

# 14.2　数组的类型

C#提供了两种类型的数组：

- 一维数组可以认为是单行元素或元素向量。
- 多维数组是由主向量中的位置组成的，每一个位置本身又是一个数组，称为子数组（sub-array）。子数组向量中的位置本身又是一个子数组。

另外，有两种类型的多维度数组：矩形数组（rectangular array）和交错数组（jagged array），它们有如下特性。

- 矩形数组
  - 某个维度的所有子数组有相同长度的多维数组。
  - 不管有多少维度，总是使用一组方括号。

```
int x = myArray2[4, 6, 1]              // 一组方括号
```

- 交错数组
  - 每一个子数组都是独立数组的多维度数组。
  - 可以有不同长度的子数组。
  - 为数组的每一个维度使用一对方括号。

```
jagArray1[2][7][4]                     // 三组方括号
```

图14-2演示了C#中的各种数组。

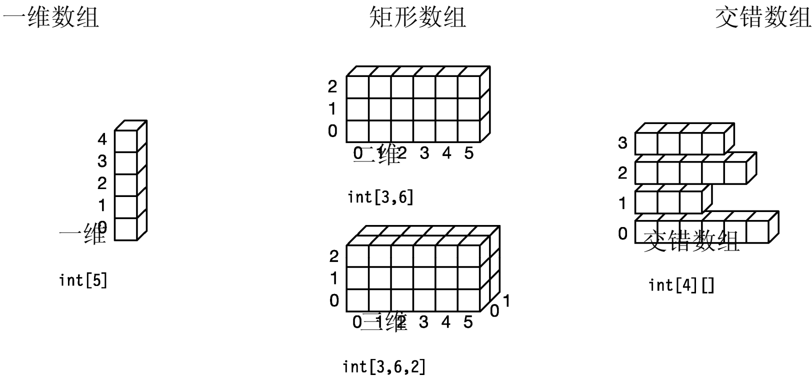


图14-2　一维、矩形以及交错数组

## 14.3　数组是对象

数组实例是从`System.Array`继承的对象。由于数组从BCL基类继承，它们也继承了很多有用的方法，如下所示。

- ❑ `Rank`：返回数组维度数的属性。
- ❑ `Length`：返回数组长度（数组中所有元素的个数）的属性。

数组是引用类型，与所有引用类型一样，有数据的引用以及数据对象本身。引用在栈或堆上，并且数组对象本身总是在堆上。图14-3演示了数组的内存配置和组成部分。

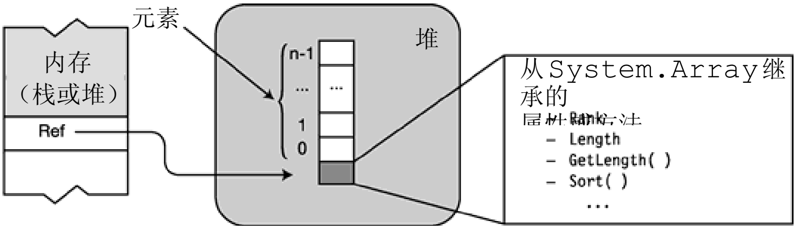


图14-3　数组的结构

尽管数组总是引用类型，但是数组的元素可以是值类型或引用类型。

- ❑ 如果存储的元素都是值类型，数组被称作值类型数组。
- ❑ 如果存储在数组中的元素都是引用类型对象，数组被称作引用类型数组。

图14-4演示了值类型数组和引用类型数组。

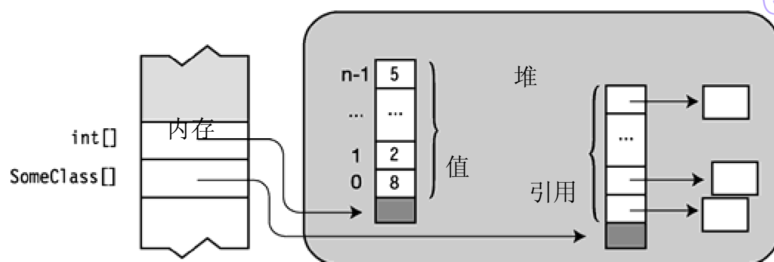



图14-4　元素可以是值引用

## 14.4　一维数组和矩形数组

一维数组和矩形数组的语法非常相似，因此我把它们放在了一起。然后，我会单独介绍交错数组。

### 声明一维数组或矩形数组

要声明一维数组或矩形数组，可以在类型和变量名称之间使用一对方括号。

方括号内的逗号就是秩说明符，它们指定了数组的维度数。秩就是逗号数量加1。比如，没有逗号代表一维数组，一个逗号代表二维数组，依此类推。

基类和秩说明符构成了数组类型。例如，如下代码行声明了`long`的一维数组。数组类型是`long[]`，读作“long数组”。

```
   秩说明符=1
      ↓
long[ ] secondArray;
  ↑
数组类型
```

如下代码展示了矩形数组声明的示例。注意：

- 可以使用任意多的秩说明符。
- 不能在数组类型区域中放数组维度长度。秩是数组类型的一部分，而维度长度不是类型的一部分。
- 数组声明后，维度数就是固定的了。然而，维度长度直到数组实例化时才会被确定。

```
  秩说明符
    ↓
int[,,]    firstArray;                    // 数组类型：三维整型数组
int[,]     arr1;                          // 数组类型：二维整型数组
long[,,]   arr3;                          // 数组类型：三维long数组
   ↑
数组类型

long[3,2,6] SecondArray;                  // 编译错误
     ↑ ↑ ↑
不允许维度长度
```

**说明**　和C/C++不同，方括号在基类型后，而不在变量名称后。

## 14.5　实例化一维数组或矩形数组

要实例化数组，我们可以使用数组创建表达式。数组创建表达式由new运算符构成，后面是基类名称和一对方括号。方块号中以逗号分隔每一个维度的长度。

下面是一维数组声明的示例：

- ❑ arr2数组是四个int的一维数组。
- ❑ mcArr数组是4个MyClass引用的一维数组。

图14-5演示了它们在内存中的布局。

```
                    四个元素
                       ↓
int[]     arr2  = new int[4];

                数组创建表达式
```

下面是矩形数组的示例，arr3数组是三维数组。

- ❑ 数组长度是3*6*2=36。
- ❑ 图14-5演示了它在内存中的布局。

```
                   维度长度
                      ↓
int[,,] arr3 = new int[3,6,2] ;
```

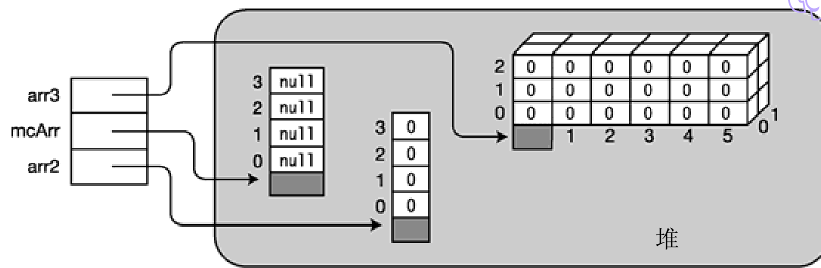



图14-5 声明和实例化数组

**说明** 与对象创建表达式不一样，数组创建表达式不包含圆括号——即使是对于引用类型数组。

## 14.6 访问数组元素

在数组中使用整型值作为索引来访问数组元素。

- 每一个维度的索引从0开始。
- 方括号内的索引在数组名称之后。

如下代码给出了声明、写入、读取一维数组和二维数组的示例：

```
int[]  intArr1 = new int[15];        // 声明一维数组
intArr1[2]     = 10;                 // 向第3个元素写入值
int var1       = intArr1[2];         // 从第3个元素读取值


int[,] intArr2 = new int[5,10];      // 声明二维数组
intArr2[2,3]   = 7;                  // 向数组写入值
int var2       = intArr2[2,3];       // 向数组读取值
```

如下代码给出了一个创建并访问一维数组的完整过程：

```
int[] myIntArray;                           // 声明数组

myIntArray = new int[4];                    // 实例化数组

for( int i=0; i<4; i++ )                    // 设置值
   myIntArray[i] = i*10;

// 读取并输出每个数组元素的值
for( int i=0; i<4; i++ )
   Console.WriteLine("Value of element {0} = {1}", i, myIntArray[i]);
```

这段代码产生了如下的输出：

```
Value of element 0 is 0
Value of element 1 is 10
Value of element 2 is 20
Value of element 3 is 30
```

## 14.7　初始化数组

当数据被创建之后，每一个元素被自动初始化为类型的默认值。对于预定义的类型，整型默认值是`0`，浮点型的默认值为`0.0`，布尔型的默认值为`false`，而引用类型的默认值则是`null`。

例如，如下代码创建了数组并将它的4个元素的值初始化为`0`。图14-6演示了内存中的布局。

```
int[] intArr = new int[4];
```

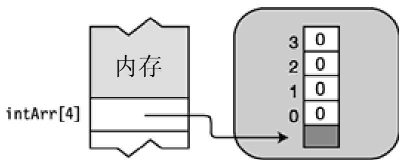



图14-6　一维数组的自动初始化

### 14.7.1　显式初始化一维数组

对于一维数组，我们可以通过在数组实例化的数组创建表达式之后包括初始化列表来设置显式初始值。

- 初始值必须以逗号分隔，并封闭在一组花括号内。
- 维度长度是可选的，因为编译器可以通过初始化值的个数来推断长度。
- 注意，在数组创建表达式和初始化列表中间没有分隔符。也就是说，没有等号或其他连接运算符。

例如，下面的代码创建了一个数组，并将它的4个元素初始化为花括号内的值。图14-7演示了内存中的布局。

```
                         初始化列表
                              ↓
int[] intArr = new int[] { 10, 20, 30, 40 };
                        ↑
                  没有连接运算符
```

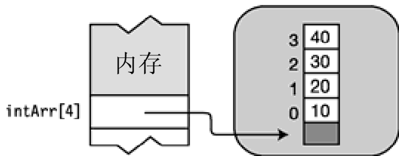



图14-7　一维数组的显式初始化

### 14.7.2　显式初始化矩形数组

要显式初始化矩形数组：

- 每一个初始值向量必须封闭在花括号内。
- 除了初始值，每一个维度的初始化列表和组成部分必须使用逗号分隔。

例如，如下代码演示了具有初始化列表的二维数组的声明。图14-8演示了在内存中的布局。

```
                                初始化列表由逗号分隔
                                     ↓        ↓
int[,] intArray2 = new int[,] { {10, 1}, {2, 10}, {11, 9} } ;
```

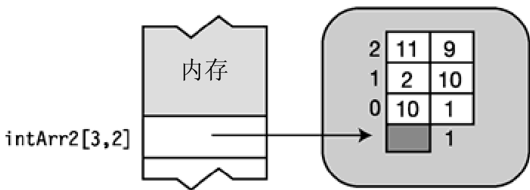



图14-8　初始化矩形数组

### 14.7.3　初始化矩形数组的语法点

矩形数组使用嵌套的、逗号分隔的初始化列表进行初始化。初始化列表嵌套在花括号内。有时会混淆，因此，对于嵌套、分组和逗号的正确使用，如下技巧可能会有用：

- 逗号用作元素和分组之间的分隔符。
- 逗号不在左花括号之间使用。
- 逗号不在右花括号之前使用。
- 从左向右读佚指示符，指定最后一个数字作为“元素”，其他数字作为“分组”。

例如，下面的声明可以读作：“`intArray`有四组两个元素一组的分组三组。”

```
                                      由逗号分隔的嵌套初始化列表
int[,,] intArray = new int[4,3,2] {          ↓          ↓          ↓
                                    { {8, 6},  {5,  2}, {12, 9} },
                                    { {6, 4},  {13, 9}, {18, 4} },
                                    { {7, 2},  {1, 13}, {9,  3} },
                                    { {4, 6},  {3,  2}, {23, 8} }
                                  };
```

### 14.7.4　快捷语法

在语句中结合声明、数组创建和初始化时，我们可以省略语法的数组创建表达式部分。快捷语法如图14-9所示。

```
int[] arr1 = new int[3] {10, 20, 30};  } Equivalent
int[] arr1 =            {10, 20, 30};  } 等价

int[,] arr = new int[2,3] {{0, 1, 2}, {10, 11, 12}};  } Equivalent
int[,] arr =              {{0, 1, 2}, {10, 11, 12}};  } 等价
```

图14-9　声明、创建以及初始化数组的快捷语法

## 14.7.5　隐式类型数组

直到现在，我们一直都在数组声明的开始处显式指定数组类型。然而，在C# 3.0中，和其他局部变量一样，数组可以是隐式类型的。也就是说：

- 当初始化数组时，我们可以让编译器根据初始化器的类型来推断数组类型。只要所有初始化器能隐式转换为单个类型，就可以这么做。
- 和隐式类型的局部变量一样，使用var关键字来替代数组类型。

如下代码演示了三组数组的声明的显式版本和隐式版本。第一组是一维int数组。第二组是二维int数组。第三组是字符串数组。注意，在隐式类型intArr4的声明中，我们仍然需要在初始化中提供秩说明符。

```
 显式                      显式
  ↓                         ↓
int [] intArr1 = new int[] { 10, 20, 30, 40 };
var    intArr2 = new    [] { 10, 20, 30, 40 };
 ↑                         ↑
关键字                     推断

int[,] intArr3 = new int[,] { { 10, 1 }, { 2, 10 }, { 11, 9 } };
var    intArr4 = new    [,] { { 10, 1 }, { 2, 10 }, { 11, 9 } };
                          ↑
                        秩说明符

string[] sArr1 = new string[] { "life", "liberty", "pursuit of happiness" };
var      sArr2 = new       [] { "life", "liberty", "pursuit of happiness" };
```

## 14.7.6　综合内容

如下代码把我们迄今学到的知识点放在了一起。它创建、初始化并使用了一个矩形数组。

```
// 声明、创建和初始化一个隐式类型的数组
var arr = new int[,] {{0, 1, 2}, {10, 11, 12}};

// 输出值
for( int i=0; i<2; i++ )
   for( int j=0; j<3; j++ )
      Console.WriteLine("Element [{0},{1}] is {2}", i, j, arr[i,j]);
```

这段代码产生了如下的输出：

```
Element [0,0] is 0
Element [0,1] is 1
Element [0,2] is 2
Element [1,0] is 10
Element [1,1] is 11
Element [1,2] is 12
```

## 14.8 交错数组

交错数组是数组的数组。与矩形数组不同，交错数组的子数组可以有不同数目的元素。例如，如下代码声明了一个二维交错数组。图14-10演示了数组在内存中的布局。

- 第一个维度的长度是3。
- 声明可以读作“jagArr是三个int数组的数组”。
- 注意，图中有四个数组对象——其中一个针对顶层数组，另外三个针对子数组。

```
int[][] jagArr = new int[3][];    // 声明并创建顶层数组
        ...                       // 声明并创建子数组
```

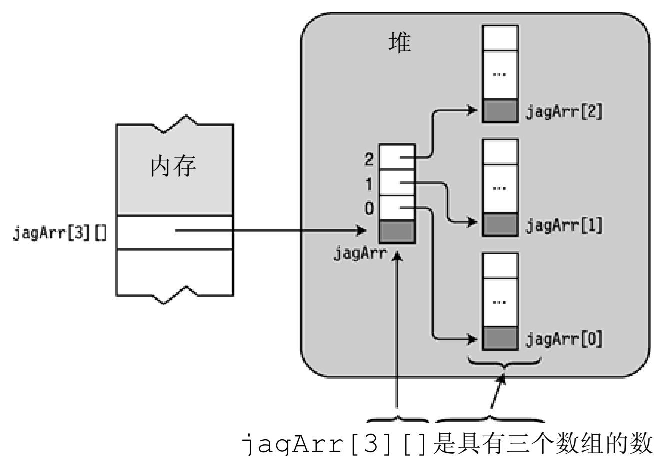



图14-10 交错数组是数组的数组

### 14.8.1 声明交错数组

交错数组的声明语法要求每一个维度都有一对独立的方括号。数组变量声明中的方括号数决定了数组的佚。

- 交错数组可以有任何大于1的维度。
- 和矩形数组一样，维度长度不能包括在数组类型的声明部分中。

```
秩说明符
  ↓
int[][]   SomeArr;             // 秩等于2
```

```
int[][][] OtherArr;          // 秩等于3
   ↑          ↑
 数组类型    数组名
```

## 14.8.2 快捷实例化

我们可以将用数组创建表达式创建的最高级别数组和交错数组的声明相结合，如下面的声明所示。结果如图14-11所示。

```
                   三个子数组
                      ↓
int[][] jagArr = new int[3][];
```

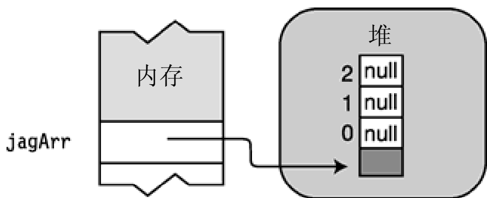



图14-11 快捷最高级别实例化

不能在声明语句中初始化最高级别数组之外的数组。

```
                       允许
                        ↓
int[][] jagArr = new int[3][4];            // 编译错误
                           ↑
                          不允许
```

## 14.8.3 实例化交错数组

和其他类型的数组不一样，交错数组的完全初始化不能在一个步骤中完成。由于交错数组是独立数组的数组——每一个数组必须独立创建。实例化完整的交错数组需要如下步骤：

(1) 首先，实例化顶层数组。

(2) 其次，分别实例化每一个子数组，把新建数组的引用赋值给包含它们的数组的合适元素。

例如，如下代码演示了二维交错数组的声明、实例化和初始化。注意，在代码中，每一个子数组的引用都赋值给了顶层数组的元素。步骤1到步骤4与图14-12中被编号的表示相对应。

```
int[][] Arr = new int[3][];                  // 1. 实例化顶层数组

Arr[0] = new int[] {10, 20, 30};             // 2. 实例化子数组
Arr[1] = new int[] {40, 50, 60, 70};         // 3. 实例化子数组
Arr[2] = new int[] {80, 90, 100, 110, 120};  // 4. 实例化子数组
```

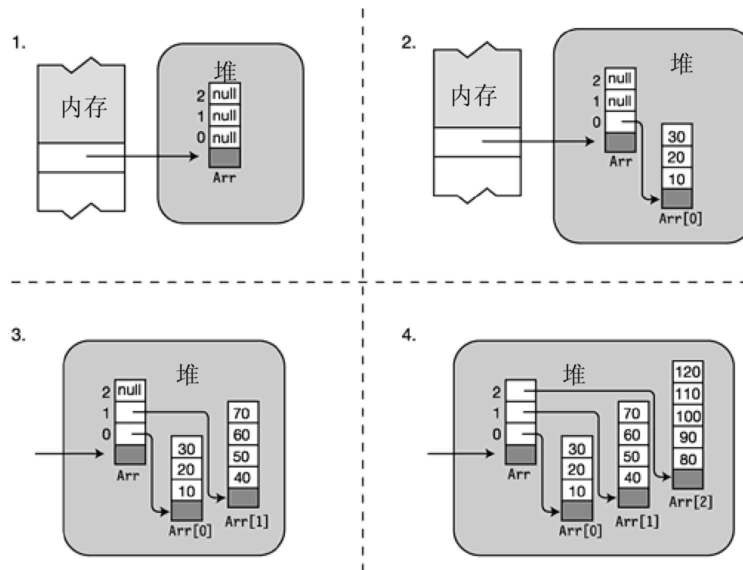



图14-12　创建一个二维交错数组

## 14.8.4　交错数组中的子数组

由于交错数组中的子数组本身就是数组，因此交错数组中也可能有矩形数组。例如，如下代码创建了一个有三个二维矩形数组的交错数组，并将它们初始化，然后显示了它们的值。

图14-13演示了结构。

代码使用了数组的继承自`System.Array`的`GetLength(int n)`方法来获取数组中指定维度的长度。

```
int[][,] Arr;          // 带有二维数组的交错数组
Arr = new int[3][,];   // 实例化带有三个二维数组的交错数组

Arr[0] = new int[,] { { 10, 20 },          { 100, 200 }           };
Arr[1] = new int[,] { { 30, 40, 50 },      { 300, 400, 500 }      };
Arr[2] = new int[,] { { 60, 70, 80, 90 }, { 600, 700, 800, 900 } };

                                    ↓ 获取Arr维度0的长度
for (int i = 0; i < Arr.GetLength(0); i++)
{                                       ↓ 获取Arr[i]维度0的长度
   for (int j = 0; j < Arr[i].GetLength(0); j++)
   {                                      ↓ 获取Arr[i]维度1的长度
      for (int k = 0; k < Arr[i].GetLength(1); k++) {
         Console.WriteLine
                  ("[{0}][{1},{2}] = {3}", i, j, k, Arr[i][j, k]);
      }
      Console.WriteLine("");
   }
   Console.WriteLine("");
}
```

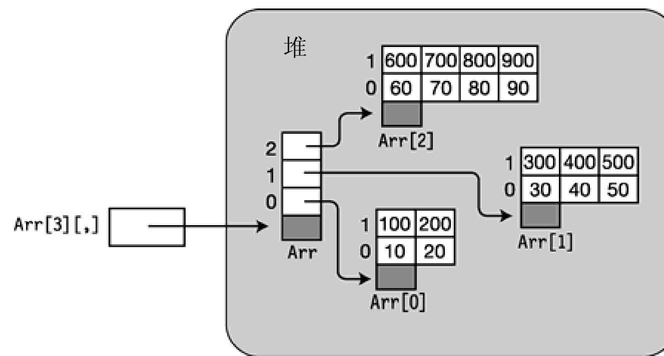



图14-13 三个二维数组构成的交错数组

## 14.9 比较矩形数组和交错数组

矩形数组和交错数组的结构的区别非常大。例如，图14-14演示了3×3的矩形数组以及一个由三个长度为3的一维数组构成的交错数组的结构。

- 两个数组都保存了9个整数，但是它们的结构却很不相同。
- 矩形数组只有单个数组对象，而交错数组有4个数组对象。

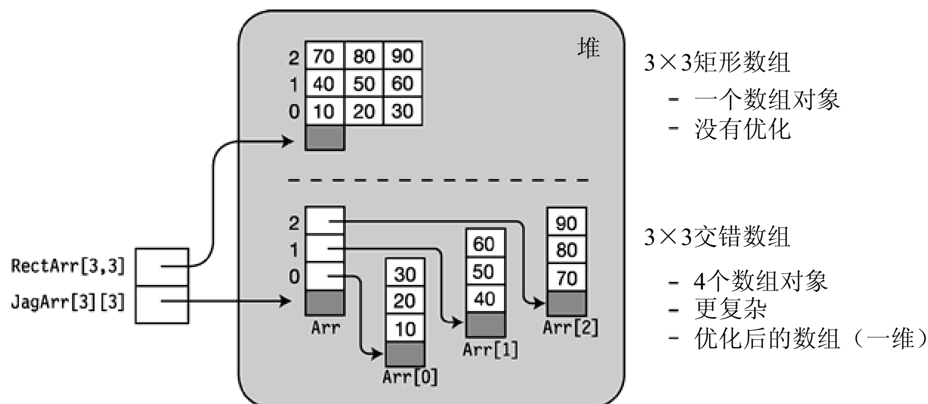



图14-14 比较矩形数组和交错数组的结构

在CIL中，一维数组有特定的指令用于性能优化。矩形数组没有这些指令，并且不在相同级别进行优化。因此，有时使用一维数组（可以被优化）的交错数组相比矩形数组会更有效率。

另一方面，矩形数组的编程复杂度更小，因为它会被作为一个单元而不是数组的数组。

## 14.10 foreach语句

foreach语句允许我们连续访问数组中的每一个元素。其实它是一个比较普遍的结构，因为可以和其他集合类型一起使用——但是在这部分内容中，我们只会讨论它和数组的使用，第20章会介绍它和其他集合类型的使用。

有关foreach的重点如下所示。

- 迭代变量是临时的，只读的，并且和数组中元素的类型相同。foreach语句使用迭代变量来连续表示数组中的每一个元素。
- foreach语句的语法如下，其中：
  - Type是数组中元素的类型。我们可以显式提供它的类型，或者，从C# 3.0开始，也可以隐式提供它的类型并通过编译器来推断，因为编译器知道数组的类型。
  - 标识符是迭代变量的名字。
  - 数组名称是要处理的数组的名字。
  - 语句是要为数组中的每一个元素执行一次的单条语句或语句块。

```
        显示类型迭代变量声明
              ↓
foreach( Type Identifier in ArrayName )
    Statement
        隐式类型迭代变量声明
              ↓
foreach( var Identifier in ArrayName )
    Statement
```

在之后的内容中，有时会使用隐式类型，而有时又会使用显式类型，这样我们就可以看到使用的确切类型，但是两种形式的语法是等价的。

foreach语句以如下方式工作：

- 从数组的第一个元素开始并把它赋值给迭代变量。
- 它然后执行语句主体。在主体中，我们可以把迭代变量作为数组元素的只读别名。
- 在主体执行之后，foreach语句选择数组中的下一个元素并重复处理。

这样，它就循环遍历了数组，允许我们逐个访问每一个元素。例如，如下代码演示了foreach语句和一个具有4个整数的一维数组的使用：

- foreach语句的主体WriteLine为数组的每一个元素执行一次。
- 第一次遍历时，迭代变量item就有数组的第一个元素的值。每次成功执行后，它就有了数组中下一个元素的值。

```
int[] arr1 = {10, 11, 12, 13};
      迭代变量声明
                              使用迭代变量
          ↓
foreach( int item in arr1 )         ↓
   Console.WriteLine("Item Value: {0}", item);
```

## 14.10.1　迭代变量是只读的

由于迭代变量的值是只读的，所以它不能被改变。但是，对于值类型数组和引用类型数组而言效果不一样。

对于值类型数组，我们不能改变数组的数据。例如，在如下的代码中，尝试改变迭代变量中

的数据产生了编译时错误消息：

```
int[] arr1 = {10, 11, 12, 13};

foreach( int item in arr1 )
   item++;     // Compilation error. Changing variable value is not allowed.
```

对于引用类型，我们仍然不能改变迭代变量，但是迭代变量只是保存了数据的引用，而不是数据本身。因此，我们可以通过迭代变量改变数据。

如下代码创建了一个有4个MyClass对象的数组并将其初始化。在第一个foreach语句中，每一个对象中的数据被改变。在第二个foreach语句中，从对象读取改变后的值。

```
class MyClass
{
   public int MyField = 0;
}

class Program  {
   static void Main() {
      MyClass[] mcArray = new MyClass[4];             //创建数组
      for (int i = 0; i < 4; i++)
      {
         mcArray[i] = new MyClass();                  //创建类对象
         mcArray[i].MyField = i;                      //设置字段
      }
      foreach (MyClass item in mcArray)
         item.MyField += 10;                          //改变数据

      foreach (MyClass item in mcArray)
         Console.WriteLine("{0}", item.MyField);      //读取改变的数据
   }
}
```

这段代码产生了如下的输出：

```
10
11
12
13
```

## 14.10.2 foreach语句和多维数组

在多维数组中，元素的处理次序是最右边的索引号最先递增。当索引从0到长度减1时，下一个左边的索引被递增，右边的索引被重置成0。

### 1. 矩形数组的示例

如下代码演示了foreach语句用于矩形数组：

```
class Program
{
   static void Main()
   {
      int total = 0;
      int[,] arr1 = { {10, 11}, {12, 13} };

      foreach( var element in arr1 )
      {
         total += element;
         Console.WriteLine
                  ("Element: {0}, Current Total: {1}", element, total);
      }
   }
}
```

这段代码产生了如下的输出：

```
Element: 10, Current Total: 10
Element: 11, Current Total: 21
Element: 12, Current Total: 33
Element: 13, Current Total: 46
```

### 2. 交错数组的示例

一个交错数组是数组的数组，我们必须为交错数组中的每一个维度使用独立的foreach语句。foreach语句必须嵌套以确保每一个嵌套数组都被正确处理。

例如，在如下代码中，第一个foreach语句遍历了顶层数组（arr1）选择了下一个要处理的子数组。内部的foreach语句处理了子数组的每一个元素。

```
class Program
{
   static void Main( )
   {
      int total = 0;
```

```
        int[][] arr1 = new int[2][];
        arr1[0] = new int[] { 10, 11 };
        arr1[1] = new int[] { 12, 13, 14 };

        foreach (int[] array in arr1)       // 处理顶层数组
        {
           Console.WriteLine("Starting new array");
           foreach (int item in array)      // 处理第二层数组
           {
              total += item;
              Console.WriteLine("  Item: {0}, Current Total: {1}", item, total);
           }
        }
     }
  }
```

这段代码产生了如下的输出：

```
Starting new array
  Item: 10, Current Total: 10
  Item: 11, Current Total: 21
Starting new array
  Item: 12, Current Total: 33
  Item: 13, Current Total: 46
  Item: 14, Current Total: 60
```

## 14.11　数组协变

在某些情况下，即使某个对象不是数组的基类型，我们也可以把它赋值给数组元素。这种属性叫做协变（covariance）。在下面的情况下可以使用协变：

- 数组是引用类型数组。
- 在赋值的对象类型和数组基类型之间有隐式转换或显式转换。

由于在派生类和基类之间总是有隐式转换的，因此总是可以将一个派生类的对象赋值给为基类声明的数组。

例如，如下代码声明了两个类，A和B，B类从继承自A类。最后一行展示了把类型B的对象赋值给类型A的数组元素而产生的协变。

图14-15演示了代码的内存布局。

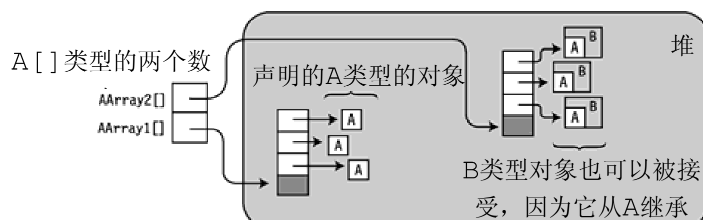



图14-15　数组出现协变

```
class A { ... }                                    // 基类
class B : A { ... }                                // 派生类

class Program {
   static void Main() {
      // 两个A[]类型的数组
      A[] AArray1 = new A[3];
      A[] AArray2 = new A[3];

      // 普通：将A类型的对象赋值给A类型的数组
      AArray1[0] = new A(); AArray1[1] = new A(); AArray1[2] = new A();

      // 协变：将B类型的对象赋值给A类型的数组
      AArray2[0] = new B(); AArray2[1] = new B(); AArray2[2] = new B();
   }
}
```

**说明**　值类型数组没有协变。

## 14.12　数组继承的有用成员

我之前提到过，C#数组从System.Array类继承。它们可以从基类继承很多有用的属性和方法，表14-1列出了其中最有用的一些。

表14-1　数组继承的一些有用成员

| 成　员 | 类　型 | 生存期 | 意　义 |
|---|---|---|---|
| Rank | 属性 | 实例 | 获取数组的维度数 |
| Length | 属性 | 实例 | 获取数组中所有维度的元素总和 |
| GetLength | 方法 | 实例 | 返回数组的指定维度的长度 |
| Clear | 方法 | 静态 | 设置元素的范围为0或null |
| Sort | 方法 | 静态 | 在一维数组中对元素进行排序 |
| BinarySearch | 方法 | 静态 | 使用二进制搜索，搜索一维数组中的值 |
| Clone | 方法 | 实例 | 进行数组的浅复制——复制值类型数组和引用类型数组的元素 |
| IndexOf | 方法 | 静态 | 返回一维数组中遇到的第一个值 |
| Reverse | 方法 | 静态 | 将一维数组中的某一范围内的元素顺序倒过来 |
| GerUpperBound | 方法 | 实例 | 获取指定维度的上限 |

例如，下面的代码使用了其中的一些属性和方法：

```
public static void PrintArray(int[] a)
{
   foreach (var x in a)
      Console.Write("{0}  ", x);
   Console.WriteLine("");
}

static void Main()
{
   int[] arr = new int[] { 15, 20, 5, 25, 10 };  PrintArray(arr);
   Array.Sort(arr);                              PrintArray(arr);
   Array.Reverse(arr);                           PrintArray(arr);

   Console.WriteLine();
   Console.WriteLine("Rank = {0}, Length = {1}",arr.Rank, arr.Length);
   Console.WriteLine("GetLength(0)      = {0}",arr.GetLength(0));
   Console.WriteLine("GetType()         = {0}",arr.GetType());
}
```

这段代码产生了如下的输出：

```
15   20   5   25   10
5   10   15   20   25
25   20   15   10   5

Rank = 1, Length = 5
GetLength(0)      = 5
GetType()         = System.Int32[]
```

## Clone方法

Clone方法为数组进行浅复制。也就是说，它只创建了数组本身的克隆。如果是引用类型数组，它不会复制元素引用的对象。对于值类型数组和引用类型数组而言，有不同的结果。

- 克隆值类型数组会产生两个独立数组。
- 克隆引用类型数组会产生指向相同对象的两个数组。

Clone方法返回object类型的引用，它必须被强制转换成数组类型。

```
int[] intArr1 = { 1, 2, 3 };
                  数组类型          返回
                     ↓               ↓
int[] intArr2 = ( int[] ) intArr1.Clone();
```

例如，如下代码给出了一个克隆值类型数组的示例，它产生了两个独立的数组。图14-16演示了代码中的一些步骤。

```
static void Main()
{
   int[] intArr1 = { 1, 2, 3 };                          // 步骤1
   int[] intArr2 = (int[]) intArr1.Clone();              // 步骤2

   intArr2[0] = 100; intArr2[1] = 200; intArr2[2] = 300;  // 步骤3
}
```

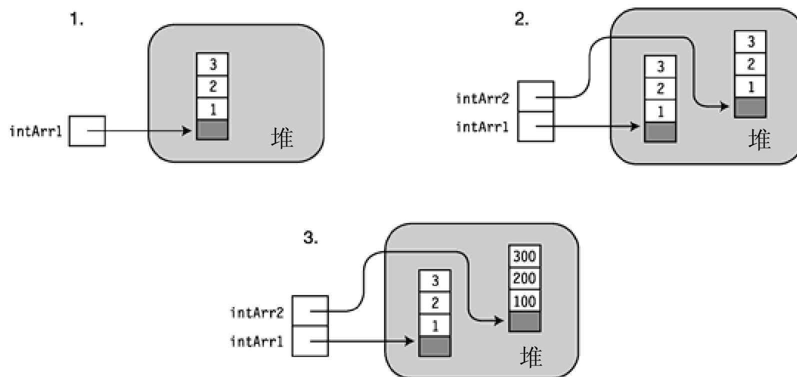



图14-16　克隆值类型数组产生了两个独立数组

克隆引用类型数组会产生指向相同对象的两个数组，如下代码给出了这个示例。图14-17演示了代码中的一些步骤。

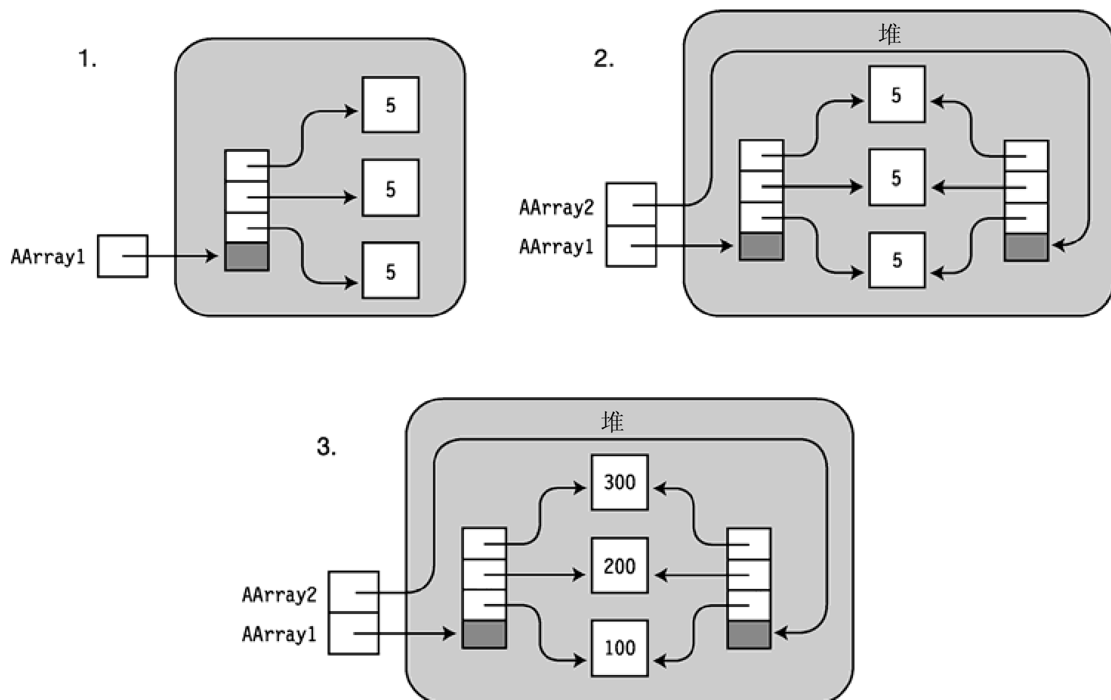



图14-17　克隆引用类型数组产生了引用相同对象的两个数组

```
class A
{
   public int Value = 5;
}

class Program
{
   static void Main()
   {
      A[] AArray1 = new A[3] { new A(), new A(), new A() };      // 步骤1
      A[] AArray2 = (A[]) AArray1.Clone();                       // 步骤2

      AArray2[0].Value = 100;
      AArray2[1].Value = 200;
      AArray2[2].Value = 300;                                    // 步骤3
   }
}
```

## 14.13 比较数组类型

表14-2总结了三种类型的数组的重要相似点和不同点。

表14-2 比较数组类型的总结

| 数组类型 | 数组对象 | 语法 | | 形状 |
|---|---|---|---|---|
| | | 结构 | 逗号 | |
| 一维<br>● 在CIL中优化指令 | 1 | 单组 | 没有 | 一维int[3] |
| 矩形<br>● 多维度<br>● 多维数组中的子数组必须是相同长度的 | 1 | 单组 | 有 | 二维int[3,6]<br>三维int[3,6,2] |
| 交错<br>● 多维度<br>● 子数组可以有不同长度 | 多个 | 多组 | 没有 | 交错int[4][] |

第 15 章

# 委　　托

15

**本章内容**

- ❑ 什么是委托
- ❑ 声明委托类型
- ❑ 创建委托对象
- ❑ 赋值委托
- ❑ 组合委托
- ❑ 为委托增加方法
- ❑ 从委托移除方法
- ❑ 调用委托
- ❑ 委托的示例
- ❑ 调用带返回值的委托
- ❑ 调用带引用参数的委托
- ❑ 匿名方法
- ❑ Lambda表达式

## 15.1　什么是委托

委托（delegate）可以认为是这样的对象，它包含具有相同签名和返回值类型的有序方法列表。

- ❑ 方法的列表称为调用列表（invocation list）。
- ❑ 当委托被调用时，它调用列表中的每一个方法。

图15-1表示一个调用列表中有4个方法的委托。

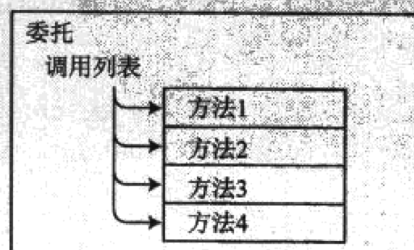



图15-1　委托作为方法的列表

包含单个方法的委托和C++的函数指针相似。然而，与函数指针不同的是，委托是面向对象的并且是类型安全（type-safe）的。

### 调用列表中的方法

由委托保存的方法可以来自任何类或结构，只要它们同时匹配委托的如下两点：

- 返回值
- 签名（包括ref和out修饰符）

调用列表中的方法可以是实例方法或是静态方法。

## 15.2 声明委托类型

委托是类型，就好像类是类型一样。与类一样，委托类型必须在被用来创建变量以及类型的对象之前声明。如下示例代码声明了委托类型。

委托类型声明和所有类型声明一样，不需要在类内部声明。

```
 关键词     委托类型名
   ↓           ↓
delegate void MyDel ( int x );
```

委托类型的声明看上去与方法的声明很相似，有返回类型和签名。返回类型和签名指定了委托接受的方法的形式。

例如，如下代码声明了MyDel类型的委托。声明指定了这种类型的委托只会接受不返回值并且有单个int参数的方法。图15-2的左边演示了委托类型，右边演示了委托对象。

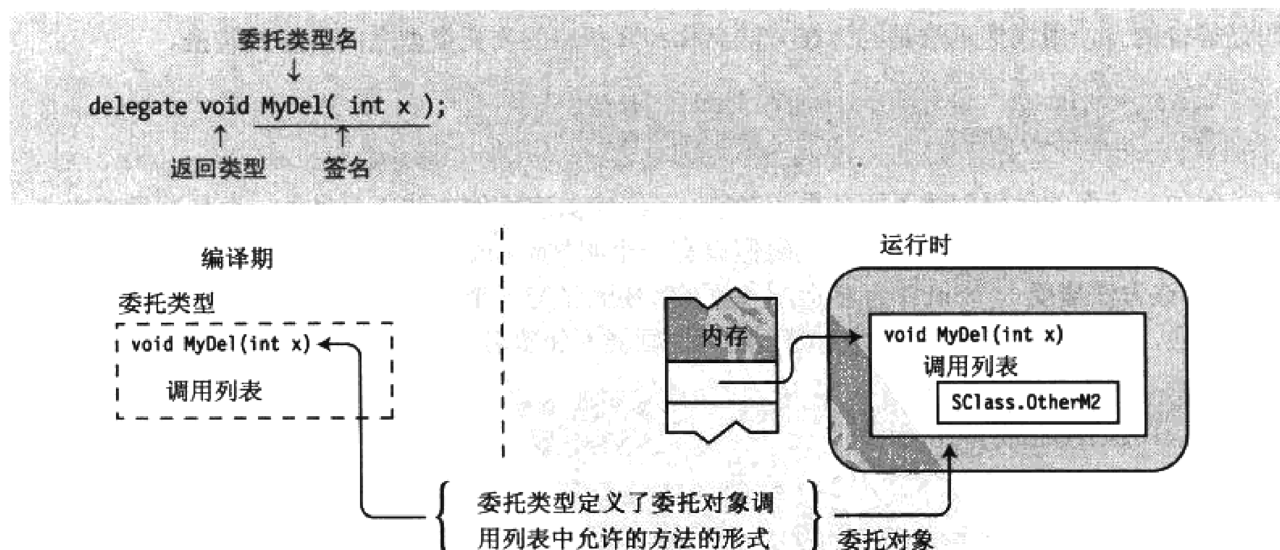



图15-2 委托类型和对象

委托类型声明在两个方面与方法声明不同。委托类型声明：

- 以delegate关键词开头。
- 没有方法主体。

## 15.3 创建委托对象

委托是引用类型，因此有引用和对象。在委托类型声明之后，我们可以声明变量并创建类型的对象。如下代码演示了委托类型的变量的声明：

```
委托类型  变量
  ↓        ↓
MyDel   delVar;
```

有两种创建委托对象的方式，第一种是使用带new运算符的对象创建表达式，如下面代码所示。new运算符的操作数的组成如下：

- 委托类型名。
- 一组圆括号，其中包含作为调用列表中第一个成员的方法的名字。方法可以是实例方法或静态方法。

```
                         实例方法
                            ↓
delVar = new MyDel( myInstObj.MyM1 );  // 创建委托并保存引用
dVar   = new MyDel( SClass.OtherM2 );  // 创建委托并保存引用
                            ↑
                         静态方法
```

我们还可以使用快捷语法，它仅由方法说明符构成，如下面代码所示。这段代码和之前的代码是等价的。使用快捷语法是因为在方法名称和其相应的委托类型之间有隐式转换。

```
delVar = myInstObj.MyM1;          // 创建委托并保存引用
dVar   = SClass.OtherM2;          // 创建委托并保存引用
```

例如，下面的代码创建了两个委托对象——一个具有实例方法，而另外一个具有静态方法。图15-3演示了委托的实例化。这段代码假设有一个叫做myInstObj的类对象，它有一个叫做MyM1的方法，该方法接受一个int作为参数，不返回值。还假设有一个名为SClass的类，它有一个OtherM2静态方法，该方法具有与MyDel委托相匹配的返回类型和签名。

```
delegate void MyDel(int x);                 // 声明委托类型
MyDel delVar, dVar;                         // 创建两个委托变量
                         实例方法
                            ↓
delVar = new MyDel( myInstObj.MyM1 );       // 创建委托并保存引用
dVar   = new MyDel( SClass.OtherM2 );       // 创建委托并保存引用
                            ↑
                         静态方法
```

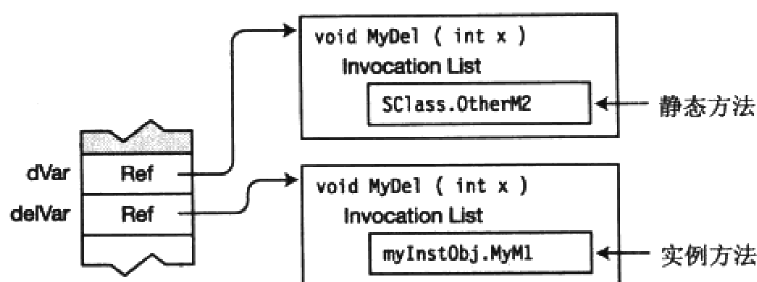



图15-3 初始化委托

除了为委托分配内存，创建委托对象还会把第一个方法放入委托的调用列表。我们还可以在同一条语句中创建变量和初始化对象。例如，下面的语句产生了与图15-3所示的相同的配置。

```
MyDel delVar = new MyDel( myInstObj.MyM1 );
MyDel dVar   = new MyDel( SClass.OtherM2 );
```

如下语句使用快捷语法，也产生了图15-3所示的结果。

```
MyDel delVar = myInstObj.MyM1;
MyDel dVar   = SClass.OtherM2;
```

## 15.4 赋值委托

由于委托是引用类型，我们可以通过给它赋值来改变包含在委托变量中的引用。旧的委托对象会被垃圾回收器回收。

例如，下面的代码设置并修改了delVar的值。图15-4演示了这段代码。

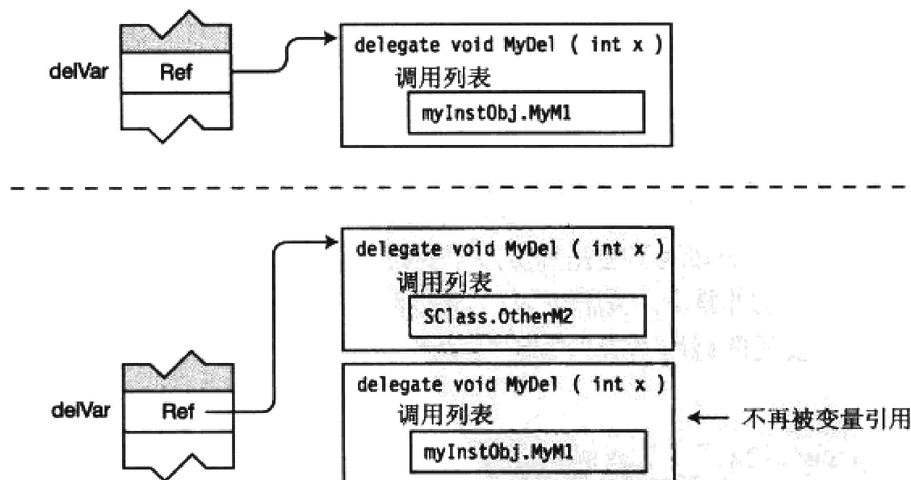



图15-4 给委托变量赋值

```
MyDel delVar;
delVar = myInstObj.MyM1;   // Create and assign the delegate object.
```

```
    ...
delVar = SClass.OtherM2;    // Create and assign the new delegate object.
```

## 15.5 组合委托

迄今为止，我们见过的所有委托在调用列表中都只有一个方法。委托可以使用额外的运算符来“组合”。这个运算最终会创建一个新的委托，其调用列表是两个操作数的委托调用列表的副本的连接。

例如，如下代码创建了3个委托。第三个委托由前两个委托组合。

```
MyDel delA = myInstObj.MyM1;
MyDel delB = SClass.OtherM2;

MyDel delC = delA + delB;                    // 组合调用列表
```

尽管术语组合委托（combining delegate）让我们觉得好像操作数委托被修改了，其实它们并没有被修改，委托是恒定的。委托对象被创建后不会再被改变。

图15-5演示了之前代码的结果。注意，操作数委托没有被改变。

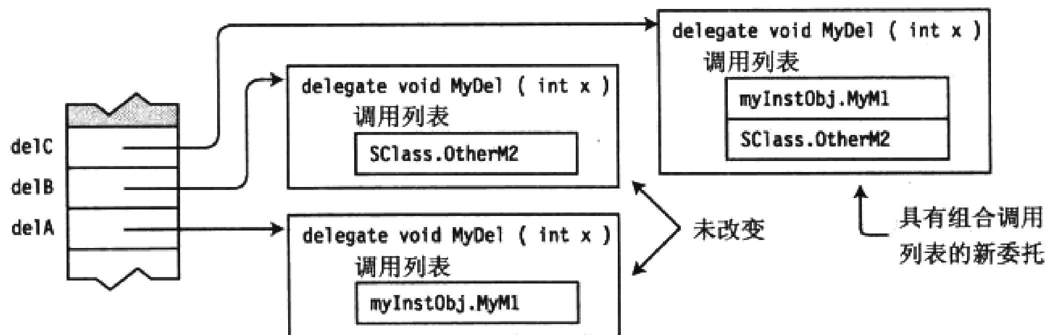



图15-5 组合委托

## 15.6 为委托增加方法

尽管通过之前的内容我们知道了委托其实是不变的，C#提供了看上去可以为委托增加方法的语法，以这种方式考虑，它非常棒。我们可以通过使用+=运算符来为委托增加方法或另一个委托。

例如，如下代码为委托的调用列表“增加”了两个方法。方法加在了调用列表的底部。图15-6演示了结果。

```
MyDel delVar  = inst.MyM1;      // 创建并初始化
delVar       += SCl.m3;         // 增加方法
delVar       += X.Act;          // 增加方法
```

当然，在使用+=运算符时，实际发生的是创建了一个新的委托，其调用列表是左边的委托加上右边方法的组合。

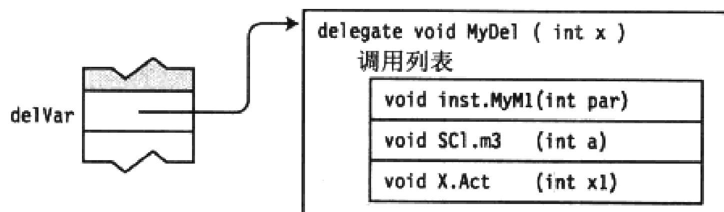



图15-6 为委托增加方法的结果

## 15.7 从委托移除方法

我们还可以使用-=运算符从委托移除方法。如下代码演示了-=运算符的使用。图15-7演示了这段代码应用在图15-7演示的委托上的结果。

```
delVar -= SCl.m3;              // Remove the method from the delegate.
```

与为委托增加方法一样，其实是创建了一个新的委托。新的委托是旧委托的副本——只是没有了已经被移除方法的引用。

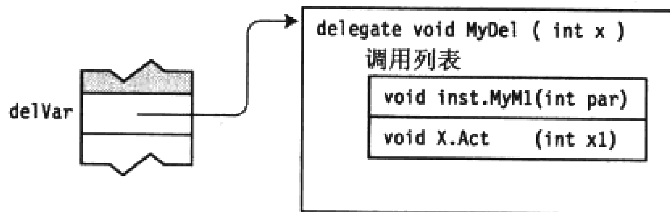



图15-7 从委托移除代码的结果

如下是移除委托时需要记住的一些事项：

- 如果在调用列表中的方法有多个实例，-=运算符将从列表最后开始搜索，并且移除第一个与方法匹配的实例。
- 试图删除委托中不存在的方法没有效果。
- 试图调用空委托会抛出异常。
- 我们可以通过把委托和null进行比较来判断委托的调用列表是否为空。如果调用列表为空，则委托是null。

## 15.8 调用委托

可以像调用方法一样简单地调用委托。用于调用委托的参数将会用于调用调用列表中的每一个方法（除非有一个参数是输出参数，我们稍后会介绍）。

例如，如下代码中的delVar委托接受了一个int型输入值。使用参数调用委托就会使用相同的参数值（在这里是55）调用它的调用列表中的每一个成员。图15-8演示了调用。

```
MyDel delVar  = inst.MyM1;
delVar       += SCl.m3;
delVar       += X.Act;
   ...
delVar( 55 );                        // 调用委托
   ...
```

一个方法可以在调用列表中出现多次。如果这样，当委托被调用时，每次在列表中遇到这个

方法时它都会被调用一次。

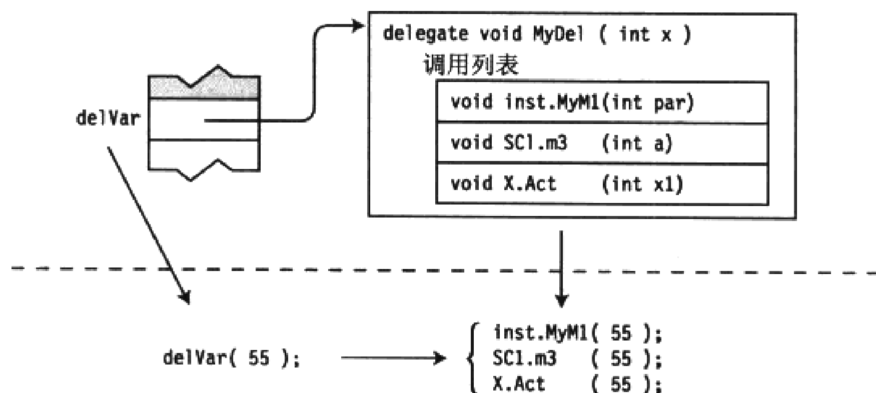



图15-8　当委托被调时，它使用被调用时的相同参数来调用调用列表中的每一个方法

## 15.9　委托的示例

如下代码定义并使用了没有参数和返回值的委托。有关代码的注意事项如下：

- Test类定义了两个打印函数。
- Main方法创建了委托的实例并增加了另外三个方法。
- 程序然后调用了委托，也就调用了它的方法。然而在调用委托之前，它检测以确保它不是null。

```
// 定义一个没有返回值和参数的委托类型
delegate void PrintFunction();

class Test
{
    public void Print1()
    { Console.WriteLine("Print1 -- instance"); }

    public static void Print2()
    { Console.WriteLine("Print2 -- static"); }
}

class Program
{
    static void Main()
    {
        Test t = new Test();    // 创建一个测试类实例
        PrintFunction pf;       // 创建一个空委托

        pf = t.Print1;          // 实例化并初始化该委托

        // Add three more methods to the delegate.
        pf += Test.Print2;
```

```
        pf += t.Print1;
        pf += Test.Print2;
        // The delegate now contains four methods.

        if( null != pf )            // 确认委托有方法
           pf();                    // 调用委托
        else
           Console.WriteLine("Delegate is empty");
     }
  }
```

这段代码产生了如下的输出：

```
Print1 -- instance
Print2 -- static
Print1 -- instance
Print2 -- static
```

## 15.10 调用带返回值的委托

如果委托有返回值并且在调用列表中有一个以上的方法，会发生下面的情况：

- 调用列表中最后一个方法返回的值就是委托调用返回的值。
- 调用列表中所有其他方法的返回值都会被忽略。

例如，如下代码声明了返回int值的委托。Main创建了委托对象并增加了另外两个方法。然后，它在WriteLine语句中调用委托并打印了它的返回值。图15-9演示了代码的图形表示。

```
delegate int MyDel( );                          // 声明有返回值的方法
class MyClass
{
   int IntValue = 5;
   public int Add2() { IntValue += 2; return IntValue;}
   public int Add3() { IntValue += 3; return IntValue;}
}

class Program
{
   static void Main( )
   {
      MyClass mc = new MyClass();
      MyDel mDel = mc.Add2;              // 创建并初始化委托
      mDel += mc.Add3;                   // 增加方法
      mDel += mc.Add2;                   // 增加方法
      Console.WriteLine("Value: {0}", mDel () );
   }                                          ↑
}                                   调用委托并使用返回值
```

这段代码产生了如下的输出：

```
Value: 12
```

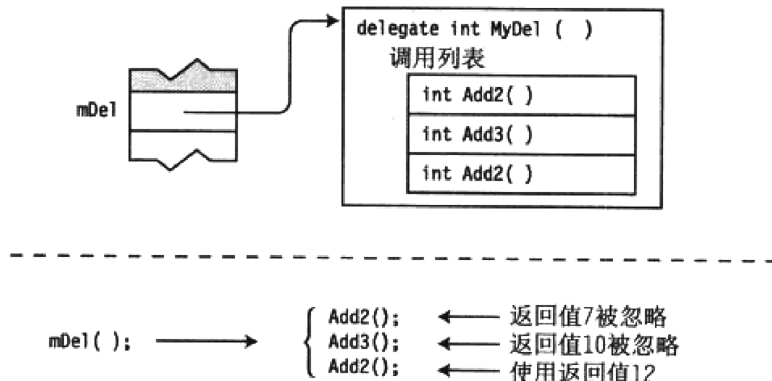



图15-9　最后一个方法执行的返回值是委托返回的值

## 15.11　调用带引用参数的委托

如果委托有引用参数，参数值会根据调用列表中的一个或多个方法的返回值而改变。在调用委托列表中的下一个方法时，参数的新值（不是初始值）会传给下一个方法。例如，如下代码调用了具有引用参数的委托。图15-10演示了这段代码。

```
delegate void MyDel( ref int X );

class MyClass
{
   public void Add2(ref int x) { x += 2; }
   public void Add3(ref int x) { x += 3; }
   static void Main()
   {
      MyClass mc = new MyClass();

      MyDel mDel = mc.Add2;
      mDel += mc.Add3;
      mDel += mc.Add2;

      int x = 5;
      mDel(ref x);

      Console.WriteLine("Value: {0}", x);
   }
}
```

这段代码产生了如下的输出：

```
Value: 12
```

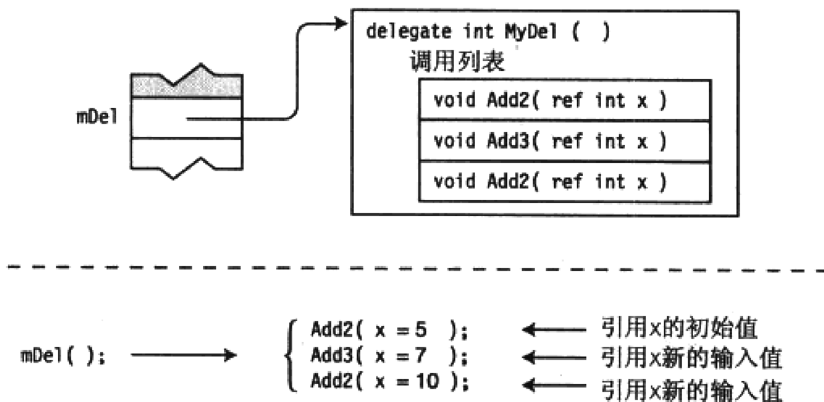



图15-10 引用参数的值会在调用间发生改变

## 15.12 匿名方法

至此，我们已经见过了使用静态方法或实例方法来初始化委托。对于这种情况，方法本身可以被代码的其他部分显式调用，当然，也就必须是某个类或结构的成员。

然而，如果方法只会被使用一次——用来初始化委托会怎么样呢？在这种情况下，除了创建委托语法的需要，没有必要创建独立的具名方法。匿名方法允许我们避免使用独立的具名方法。

*匿名方法*（anonymous method）是在初始化委托时内联（inline）声明的方法。

例如，图15-11 演示了同一个类的两个版本。左边的版本声明并使用了一个名称为Add20的方法。右边的版本使用了匿名方法来替代。没有底色的代码部分对于两个版本是一样的。

```
class Program
{
   public static int Add20(int x)
   {
      return x + 20;
   }

   delegate int OtherDel(int InParam);
   static void Main()
   {
      OtherDel del = Add20;


      Console.WriteLine("{0}", del(5));
      Console.WriteLine("{0}", del(6));
   }
}
```

命名方法

```
class Program
{



   delegate int OtherDel(int InParam);
   static void Main()
   {
      OtherDel del = delegate(int x)
                     {
                        return x + 20;
                     };
      Console.WriteLine("{0}", del(5));
      Console.WriteLine("{0}", del(6));
   }
}
```

匿名方法

图15-11 比较具名方法和匿名方法

图15-11中的两组代码都产生了如下的输出：

```
25
26
```

### 15.12.1 使用匿名方法

我们可以在如下地方使用匿名方法：

- 声明委托变量时作为初始化表达式。
- 组合委托时在赋值语句的右边。
- 为委托增加事件时在赋值语句的右边。第16章会介绍事件。

### 15.12.2 匿名方法的语法

匿名方法表达式的语法包含如下组成部分：

- delegate类型关键字。
- 参数列表，如果语句块没有使用任何参数则可以省略。
- 语句块，它包含了匿名方法的代码。

```
  关键词      参数列表              语句块
    ↓            ↓                    ↓
delegate ( Parameters ) { ImplementationCode }
```

#### 1. 返回类型

匿名方法不会显式声明返回值。然而，实现代码本身的行为必须通过返回一个在类型上与委托的返回类型相同的值来匹配委托的返回类型。如果委托有void类型的返回值，匿名方法就不能返回值。

例如，在如下代码中，委托的返回类型是int。匿名方法的实现代码因此也必须在代码路径中返回int。

```
    委托类型的返回类型
           ↓
delegate int OtherDel(int InParam);

static void Main()1
{
    OtherDel del = delegate(int x)
                   {
                       return x + 20 ;                // 返回一个整型值
                   };
        ...
}
```

#### 2. 参数

除了数组参数，匿名方法的参数列表必须在如下三方面匹配委托：

- 参数数量
- 参数类型
- 修饰符

我们可以通过使圆括号为空或省略圆括号来简化匿名方法的参数列表，但是仅在下面两项都为真的情况下才可以这样做。

❑ 委托的参数列表不包含任何out参数。

❑ 匿名方法不使用任何参数。

例如，如下代码声明了没有任何out参数的委托，匿名方法也没有使用任何参数。由于两个条件都满足了，我们就可以省略匿名方法的参数列表。

```
delegate void SomeDel ( int X );                  // 声明委托类型

SomeDel SDel = delegate                            // 省略的参数列表
               {
                  PrintMessage();
                  Cleanup();
               };
```

### 3. params参数

如果委托声明的参数列表包含了params参数，那么params关键字就会被匿名方法的参数列表忽略。例如，在如下代码中：

❑ 委托类型声明指定最后一个参数为params类型的参数。

❑ 然而，匿名方法参数列表忽略了params关键字。

```
                    在委托类型声明中使用parms关键词
                                  ↓
delegate void SomeDel( int X, params int[] Y);

                    在匹配的匿名方法中省略关键词
                                  ↓
SomeDel mDel = delegate (int X, int[] Y)
             {
                ...
             };
```

## 15.12.3 变量和参数的作用域

参数以及声明在匿名方法内部的局部变量的作用域限制在实现方法的主体之内，如图15-12所示。

```
delegate void MyDel( int x );
...
MyDel mDel = delegate ( int y )
             {
                int z = 10;
                Console.WriteLine("{0}, {1}", y, z);       } y和z的作用域
             };
Console.WriteLine("{0}, {1}", y, z);   // 编译错误
                              ↑
                           离开作用域
```

图15-12 变量和参数的作用域

例如，如下匿名方法定义了参数y和局部变量z。在匿名方法主体结束之后，y和z就不在作用域内了。最后一行代码将会产生编译错误。

### 1. 外部变量

与委托的命名方法不同，匿名方法可以访问它们外围作用域的局部变量和环境。

- 外围作用域的变量叫做**外部变量**（outer variable）。
- 用在匿名方法实现代码中的外部变量称为**被方法捕获**（captured）。

例如，图15-13中的代码演示了定义在匿名方法外部的变量x。然而，方法中的代码可以访问x并输出它的值。

### 2. 被捕获变量的生命周期的扩展

只要捕获方法还是委托的一部分，即使变量已经离开了作用域，被捕获的外部变量也会一直有效。

```
int x = 5;
   ...
MyDel mDel = delegate
             {
               Console.WriteLine("{0}", x);
             };
```

变量x定义在匿名方法作用域的前面

变量x可以在匿名方法作用域之内使用

使用外部变量x

图15-13　使用外部变量

例如，图15-14中的代码演示了被捕获变量的生命周期的扩展。

```
delegate void MyDel( );
static void Main()
{
   MyDel mDel;
   {
      int x = 5;
      mDel = delegate
             {
               Console.WriteLine("Value of x: {0}", x);
             };
   }
   // Console.WriteLine("Value of x: {0}", x);

   if (null != mDel)
      mDel( );
}
```

变量x被定义在外部块中，在匿名方法之外

x作用域

变量x被匿名方法捕获

变量x离开了作用域并且会导致编译错误

而x在这里使用，在匿名方法内部

图15-14　在匿名方法中捕获的变量

- 局部变量x在块中声明和初始化。
- 然后，委托mDel被匿名方法初始化，该匿名方法捕获了外部变量x。

- 块关闭时，x超出了作用域。
- 如果块关闭之后的WriteLine语句的注释被取消，就会产生编译器错误。因为它引用的x现在已经离开了作用域。
- 然而，mDel委托中的匿名方法在它的环境中保留了x，并在mDel被调用时输出了它的值。

图中的这段代码产生了如下的输出：

```
Value of x: 5
```

## 15.13　Lambda 表达式

C# 2.0引入了匿名方法，它允许我们在创建或为委托增加方法时包含小段内联代码。然而，匿名方法的语法有一点麻烦，而且需要一些编译器已经知道的信息。C# 3.0引入了lambda表达式，简化了匿名方法的语法，从而避免包含这些多余的信息。我们可能会希望使用lambda表达式来替代匿名方法。其实，如果lambda表达式被先引入，那么就不会有匿名方法。

在匿名方法的语法中，delegate关键字是有点多余，因为编译器已经知道我们在将方法赋值给委托。我们可以通过如下步骤把匿名方法转换为lambda表达式：

- 删除delegate关键字。
- 在参数列表和匿名方法主体之间放lambda运算符=>。lambda运算符读作“goes to”。

如下代码演示了这种转换。第一行演示了将匿名方法赋值给变量del。第二行演示了同样的匿名方法在被转换成lambda表达式之后，被赋值给了变量le1。

```
MyDel del = delegate(int x)    { return x + 1; } ;     // 匿名方法
MyDel le1 =         (int x) => { return x + 1; } ;     // 表达式
```

**说明**　术语lambda表达式来源于数学家Alonzo Church等人在1920年到1930年期间发明的lambda积分。lambda积分是用于表示函数的一套系统，它使用希腊字母lambda（λ）来表示无名函数。近来，诸如Lisp和其方言的函数式编程语言使用这个术语来表示可以直接用于描述函数定义的表达式，表达式不再需要有名字了。

这种简单的转换少了一些多余的东西，看上去也更简洁了，但是只省了6个字符。然而，编译器可以通过推断，允许我们更进一步简化lambda表达式，如下代码所示。

- 编译器还可以从委托的声明中知道委托参数的类型，因此lambda表达式允许我们省略类型参数，如le2的赋值代码所示。
  - 带有类型的参数列表称为显式类型。
  - 省略类型的参数列表称为隐式类型。
- 如果只有一个隐式类型参数，我们可以省略周围的圆括号，如le3的赋值代码所示。
- 最后，lambda表达式允许表达式的主体是语句块或表达式。如果语句块包含了一个返回语句，我们可以将语句块替换为return关键字后的表达式，如le4的赋值代码所示。

```
MyDel del = delegate(int x)    { return x + 1; } ;    // 匿名方法
MyDel le1 =         (int x) => { return x + 1; } ;    // Lambda 表达式
MyDel le2 =             (x) => { return x + 1; } ;    // Lambda 表达式
MyDel le3 =              x  => { return x + 1; } ;    // Lambda 表达式
MyDel le4 =              x  =>          x + 1    ;    // Lambda 表达式
```

最后一种形式的lambda表达式的字符只有原始匿名方法的1/4，更简洁，更容易理解。

如下代码演示了完整的转换。Main的第一行演示了被赋值给变量del的匿名方法。第二行演示了被转换成lambda表达式后的相同匿名方法，并赋值给变量le1。

```
delegate double MyDel(int par);

static void Main()
{
   MyDel del = delegate(int x)    { return x + 1; } ; // 匿名方法

   MyDel le1 =         (int x) => { return x + 1; } ; // Lambda 表达式
   MyDel le2 =             (x) => { return x + 1; } ;
   MyDel le3 =              x  => { return x + 1; } ;
   MyDel le4 =              x  =>          x + 1    ;

   Console.WriteLine("{0}", del (12));
   Console.WriteLine("{0}", le1 (12));  Console.WriteLine("{0}", le2 (12));
   Console.WriteLine("{0}", le3 (12));  Console.WriteLine("{0}", le4 (12));
}
```

有关lambda表达式的参数列表的要点如下：

- lambda表达式参数列表中的参数必须在参数数量、类型和位置上与委托相匹配。
- 表达式的参数列表中的参数不一定需要包含类型（如隐式类型），除非委托有ref或out参数——此时类型是必须的（如显式类型）。
- 如果只有一个参数，并且是隐式类型的，周围的圆括号可以被省略，否则它就是必须的。
- 如果没有参数，必须使用一组空的圆括号。

图15-15演示了lambda表达式的语法。

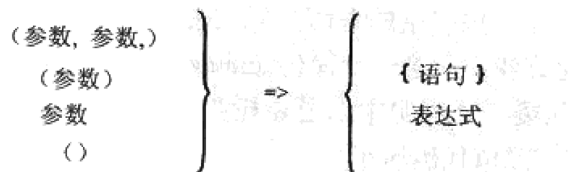



图15-15　lambda表达式的语法由lambda运算符和左边的参数部分以及右边的lambda主体构成

# 第 16 章 事件

本章内容

- 事件和委托相似
- 源代码组件概览
- 声明事件
- 触发事件
- 订阅事件
- 标准事件使用
- MyTimerClass代码
- 事件访问器

## 16.1 事件和委托相似

前面一章介绍了委托。事件的很多方面和委托相似。其实，事件就好像被简化的针对特殊用途的委托。图16-1说明了事件和委托的相似之处，注册到事件上的方法会在事件触发时被调用。

下面是一些有关事件的重要事项。

- 触发（raise）事件：调用（invoke）或触发（fire）事件的术语。当事件被触发时，所有注册到它的方法都会被依次调用。
- 发布者（publisher）：让事件被其他类或结构可见并使用的类或结构。
- 订阅者（subscriber）：把事件和发布者关联注册的类或结构。
- 事件处理程序（event handler）：注册到事件的方法。可以在事件所在的类或结构中，或者在不同的类或结构中。

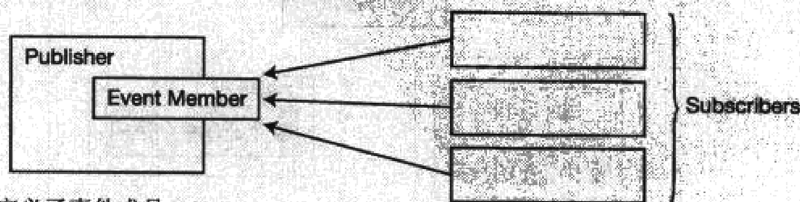



图16-1 发布者和订阅者

## 事件有私有委托

委托和事件的行为之所以相似，是有充分理由的。事件包含了一个私有的委托，如图16-2所示。有关事件的私有委托需要了解的重要事项如下：

- ❑ 事件提供了对它的私有控制委托的结构化访问。
- ❑ 与委托中的其他操作不一样，对于事件我们只可以添加、删除或调用事件处理程序。
- ❑ 事件被触发时，它调用委托来依次调用调用列表中的方法。

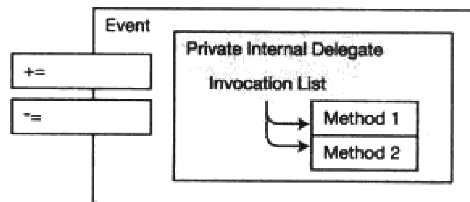


图16-2　事件是被封装的委托

注意，在图16-2中，只有+=和-=运算符在事件的左边。因为，它们是事件唯一允许的运算符。

图16-3演示了一个具有Elapsed事件的发布者类的运行时视图。右边的ClassA和ClassB，每一个都在Elapsed上注册了事件处理程序。在事件内，我们可以看到委托引用了两个事件处理程序。除了事件，发布者还包含了触发事件的代码。

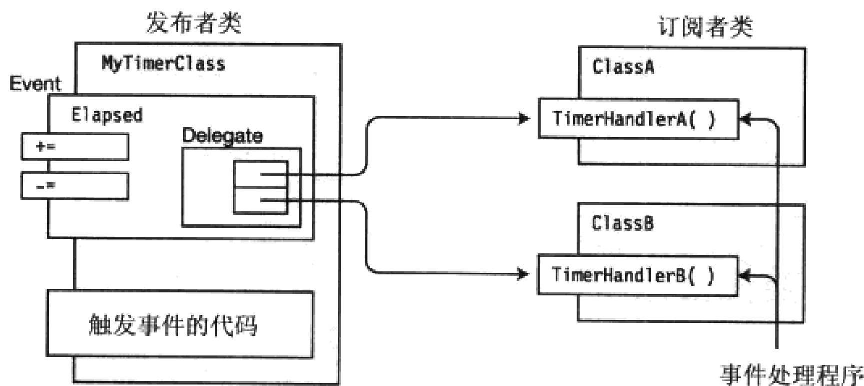


图16-3　具有一个计时器事件类的结构和术语

## 16.2　源代码组件概览

需要在事件中使用的代码有5部分。我会在下面的篇幅中依次进行介绍，它们如图16-4所示。这些组件如下所示。

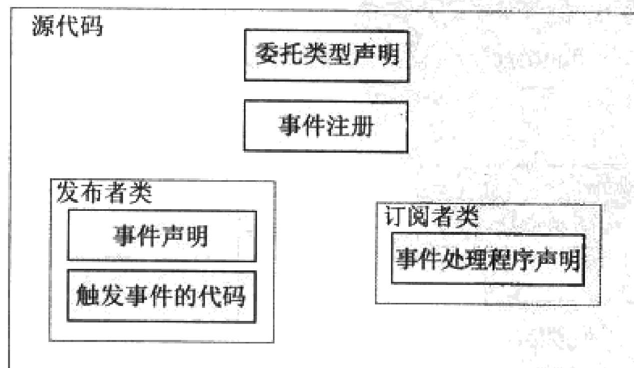


图16-4　使用事件的5个源代码组件

- 委托类型声明：事件和事件处理程序必须有共同的签名和返回类型，它们通过委托类型声明进行描述。
- 事件处理程序声明：这些在订阅者类的方法（事件处理程序）中的描述会在事件触发时被执行。它们不需要有独立的方法，它们可以是匿名方法或lambda表达式。
- 事件声明：这个事件发布者类中的声明保存并调用事件处理程序。
- 事件注册：这段代码把事件连接到事件处理程序。
- 触发事件的代码：发布者类中的这段代码调用事件导致它调用事件处理程序。

## 16.3 声明事件

发布者类必须提供事件和触发事件的代码。

创建事件比较简单——只需要委托类型和名字。事件声明的语法如下代码所示，代码中声明了一个叫做Elapsed的事件。注意如下有关Elapsed事件的内容：

- 声明在一个叫做MyTimerClass的类中。
- 它接受返回类型和签名与EventHandler委托类型匹配的事件处理程序。
- 它被声明为public，于是其他类和结构可以在这上面注册事件处理程序。

```
class MyTimerClass
{        关键字              事件名
           ↓                  ↓
   public event EventHandler Elapsed;
                     ↑
                  委托类型
```

我们可以通过使用逗号分隔的列表在一个声明语句中声明一个以上的事件。例如，下面语句声明了三个事件。

```
public event EventHandler MyEvent1, MyEvent2, OtherEvent;
                                    ↑
                                 三个事件
```

我们还可以使用static关键字让事件变成静态的，如下声明所示：

```
public static event EventHandler Elapsed;
         ↑
       关键字
```

### 16.3.1 事件是成员

一个常见的误解是把事件认为是类型，然而它不是。事件是成员，这一点引出了几个重要的特性。

- 由于事件不是类型，我们不能使用对象创建表达式（new表达式）来创建它的对象。
- 由于事件是成员：
  - 它必须声明在类或结构中，和其他成员一样；

  - 我们不能在一段可执行代码中声明事件；
- 事件成员被隐式自动初始化为null。

### 16.3.2　委托类型和EventHandler

事件声明需要委托类型的名字，我们可以声明一个委托类型或使用已存在的。如果我们声明一个委托类型，它必须指定事件保存的方法的签名和返回类型。

一个更好的方法是，使用.NET BCL使用的并被指定为事件使用标准的预定义委托类型。强烈推荐使用它，那就是EventHandler，它的声明如下代码所示。在本章之后会介绍更多关于委托EventHandler的细节。

```
public delegate void EventHandler(object sender, EventArgs e);
```

## 16.4　触发事件

事件成员本身只是保存了需要被调用的事件处理程序。如果事件没有被触发，什么都不会发生。我们需要确保在合适的时候有代码来做这件事情。

例如，如下代码触发了Elapsed事件。注意如下有关代码的事项：

- 在触发事件之前和null进行比较，从而查看是否包含任何事件处理程序，如果事件是null，则表示没有。
- 触发事件本身看起来像调用函数一样。
  - 使用事件名称，后面跟的参数列表包含在圆括号中。
  - 参数列表必须匹配事件的委托类型。

```
if (Elapsed != null)             // 确认有方法可以执行
   Elapsed(source, args);        // 抛出异常
      ↑          ↑
    事件名     参数列表
```

把事件声明和触发事件的代码放在一起便有了如下的发布者类声明。这段代码包含了两个成员：事件和一个叫做OnOneSecond的方法，它触发了该事件。

```
public class MyTimerClass
{
   public event EventHandler Elapsed;    // 声明事件

   private void OnOneSecond(object source, EventArgs args)
   {
      if (Elapsed != null)         //确认有方法可以执行
         Elapsed(source, args);
   }              ↑
               发起事件

   // 下面的代码确认OnOneSecond方法每1000毫秒被调用一次
```

```
    ...
}
```

至此，我们的OnOneSecond方法有点神秘，每1秒会被调用一次。在本章后面，我会演示如何实现它。但在这里，记住这些要点：

- 发布者类有一个作为成员的事件。
- 类包含了触发事件的代码。

## 16.5 订阅事件

要为事件添加事件处理程序，处理程序必须有和事件委托一致的返回类型和签名。

- 使用+=运算符来为事件增加事件处理程序，如下面代码所示。
- 方法可以是下面的任意一个：
  - 实例方法
  - 静态方法
  - 匿名方法
  - lambda表达式

例如，下面代码为Elapsed事件增加了三个方法：第一个是使用方法形式的实例方法，第二个是使用方法形式的静态方法，第三个是使用委托形式的实例方法。

```
类                  实例方法
↓                      ↓
mc.Elapsed += ca.TimerHandlerA;                    // 方法引用形式
mc.Elapsed += ClassB.TimerHandlerB;                // 方法引用形式
    ↑                     ↑
  事件成员              静态方法
mc.Elapsed += new EventHandler(cc.TimerHandlerC);  // 委托形式
```

和委托一样，我们可以使用匿名方法和lambda表达式来增加事件处理程序。例如，如下代码先使用lambda表达式然后使用了匿名方法。

```
mc.Elapsed += (source, args) =>                       //Lambda表达式
   {
      Console.WriteLine("Lambda expression.");
   };

mc.Elapsed += delegate(object source, EventArgs args)  //匿名方法
   {
      Console.WriteLine("Anonymous method.");
   };
```

如下程序使用了在前面定义的MyTimerClass类。代码执行如下工作：

- 它从两个不同的类实例注册两个事件处理程序。

- 在注册事件处理程序后，它休眠2秒。在这段时间内，计时器类会触发两次事件，两个事件处理器每次都会被执行。

```
public class MyTimerClass { ... }

class ClassA
{
   public void TimerHandlerA(object obj, EventArgs e)      // 事件处理程序
   {
      Console.WriteLine("Class A handler called");
   }
}

class ClassB
{
   public static void TimerHandlerB(object obj, EventArgs e)   // 静态的
   {
      Console.WriteLine("Class B handler called");
   }
}

class Program
{
   static void Main( )
   {
      ClassA ca = new ClassA();                    // 创建类对象
      MyTimerClass mc = new MyTimerClass();        // 创建计时器对象

      mc.Elapsed += ca.TimerHandlerA;              // 添加处理程序A（实例）
      mc.Elapsed += ClassB.TimerHandlerB;          // 添加处理程序B（静态的）

      Thread.Sleep(2250);
   }
}
```

这段代码和MyTimerClass放在一起产生了如下的输出：

```
Class A handler called
Class B handler called
Class A handler called
Class B handler called
```

## 移除事件处理程序

我们可以使用-=运算符从事件移除一个事件处理程序，如下所示。

```
mc.Elapsed -= ca.TimerHandlerA;          //移除事件处理程序A
```

例如，如下代码在前两次事件被触发之后移除了ClassB的事件处理程序，然后让程序再运行

2秒。

```
...
mc.Elapsed += ca.TimerHandlerA;            // 添加实例事件处理程序A
mc.Elapsed += ClassB.TimerHandlerB;        // 添加静态事件处理程序B

Thread.Sleep(2250);                        // 休眠2秒以上时间

mc.Elapsed -= ClassB.TimerHandlerB;        // 移除静态事件处理程序B
Console.WriteLine("Class B event handler removed");

Thread.Sleep(2250);                        // 休眠2秒以上时间
```

这段代码产生了如下的输出。前4行是在前2秒内两个处理程序被调用两次的结果。在ClassB的处理程序被移除后，在最后的2秒内只有ClassA的实例处理程序被调用。

```
Class A handler called
Class B handler called
Class A handler called
Class B handler called
Class B event handler removed
Class A handler called
Class A handler called
```

## 16.6 标准事件的用法

GUI编程是事件驱动的，也就是说在程序运行时，它可以在任何时候被事件打断，比如按钮点击、按下按键或系统定时器。在这些情况发生时，程序需要处理事件然后继续其他事情。

显然，对于使用C#事件而言，最好的例子是异步处理程序事件。Windows GUI编程如此广泛地使用了事件，对于事件的使用，.NET框架提供了一个强烈推荐遵循的标准模式。

事件使用的标准模式的根本就是System命名空间声明的EventHandler委托类型。EventHandler委托类型的声明如下面代码所示。

- 第一个参数用来保存触发事件的对象的引用。由于是object类型的，所以可以匹配任何类型的实例。
- 第二个参数用来保存有关状态对于应用程序来说是否合适的状态信息。
- 返回类型是void。

```
public delegate void EventHandler(object sender, EventArgs e);
```

### 16.6.1 使用EventArgs类

EventHandler委托类型的第二个参数是EventArgs类的对象，它声明在System命名空间中。你可能会想，既然第二个参数用于传递数据，EventArgs类的对象应该可以保存一些类型的数据。

你可能错了。

- EventArgs被设计为不能传递任何数据。它用于不需要传递数据的事件处理程序——通常会被忽略。
- 如果你希望传递数据，必须声明一个从EventArgs继承的类，使用合适的字段来保存需要传递的数据。

尽管EventArgs类实际上并不传递数据，但它是使用EventHandler委托模式的重要一部分。不管参数使用的实际类型是什么，object类和EventArgs总是基类。这样EventHandler就能提供一个对所有事件和事件处理器都通用的签名，只允许2个参数，而不是各自都有不同签名。

### 16.6.2　通过扩展 EventArgs 来传递数据

为了向自己的事件处理程序的第二个参数传入数据，并且又符合标准惯例，我们需要声明一个派生自EventArgs的自定义类，它可以保存我们所需传入的数据。类的名称应该以EventArgs结尾。例如，如下代码声明了一个自定义类，它能将字符串存储在名称为Message的字段中。

```
                 自定义类名     基类
                     ↓          ↓
public class MyTCEventArgs: EventArgs
{
   public string Message;                  // 存储message
   public MyTCEventArgs(string s)          // 构造函数设置message
   {
      Message = s;
   }
}s
```

### 16.6.3　使用自定义委托

既然我们已经有了一个自定义类，可以让我们在事件处理程序的第二个参数中传入数据。我们需要一个委托类型来使用新的自定义类，可以有两种方式这么做。

- 第一种方式是使用非泛型委托。实现方式如下：
  - 使用自定义的类类型创建一个新的自定义委托，如下代码所示。
  - 在事件代码的其他部分中使用新的委托名称。

```
                      自定义委托名                    自定义类
                           ↓                            ↓
public delegate void MyTCEventHandler(object sender, MyTCEventArgs e);
```

- 第二种方式是由C# 2.0引入的，使用EventHandler泛型委托的方式。C#泛型会在第19章进行介绍。要使用泛型委托，以如下方式来实现，代码如下所示。
  - 在方括号中放置自定义类。
  - 无论希望在哪里使用自定义委托类型的名称，都使用完整的字符串。例如，event声明

是这样的：

```
                 使用自定义类的泛型委托
                         ↓
public event EventHandler<MyTCEventArgs> Elapsed;
                                           ↑
                                         事件名
```

在其他4个事件相关的代码段中使用这个自定义类和自定义委托（泛型或非泛型形）。

例如，如下代码更新了MyTimerClass代码来使用叫做MyTCEventArgs的自定义EventArgs类和泛型EventHandler<>委托。

```
public class MyTCEventArgs: EventArgs                      ┐
{                                                          │
   public string Message;                                  │
                                                           │ 自定义类声明
   public MyTCEventArgs(string s) {                        │
      Message = s;                                         │
   }                                                       │
}                                                          ┘

public class MyTimerClass  泛型委托
{                              ↓
   public event EventHandler<MyTCEventArgs> Elapsed;     //事件声明

   private void OnOneSecond(object obj, EventArgs e)
   {
      if (Elapsed != null)
      {                                                            ┐
         MyTCEventArgs mtcea =                                     │
            new MyTCEventArgs("Message from OnOneSecond");         │ 发起事件的代码
         Elapsed(obj, mtcea);                                      │
      }                                                            ┘
   }
   ...
}

class ClassA
{
   public void TimerHandlerA(object obj, MyTCEventArgs e)         ┐
   {                                                               │
      Console.WriteLine("Class A Message:  {0}", e.Message);      │ 事件处理程序
   }                                                               ┘
}

class Program
{
   static void Main()
   {
```

```
        ClassA ca = new ClassA();
        MyTimerClass mc = new MyTimerClass();

        mc.Elapsed +=                                         //注册事件处理程序
           new EventHandler<MyTCEventArgs>(ca.TimerHandlerA);

        Thread.Sleep(3250);
    }
}
```

这段代码产生了如下的输出：

```
Class A Message:  Message from OnOneSecond
Class A Message:  Message from OnOneSecond
Class A Message:  Message from OnOneSecond
```

## 16.7　MyTimerClass代码

既然你已经看过了使用事件需要实现的五个组件的代码，那么我就可以展示一下代码使用的完整的MyTimerClass类。

关于类的大多数事项都很明白了——它有一个叫做Elspsed的事件可以被订阅，还有一个叫做OnOneSecond的方法每隔1秒会被调用一次并触发事件。剩下的一个问题就是："什么导致OnOneSecond每1秒就被调用一次？"

答案是：OnOneSecond本身就是一个事件处理程序，该事件处理程序订阅了System.Timers命名空间中Timer类的一个事件。Timer的事件每1 000毫秒触发一次并调用OnOneSecond事件处理程序，然后它再触发MyTimerClass类中的Elapsed事件。图16-5演示了代码的结构。

Timer类是很有用的工具，因此我会再介绍一点有关它的内容。首先，它有一个叫做Elapsed的公共事件。听上去很熟悉，因为我还用这个名字来命名了MyTimerClass中的事件。除了名字之外没有其他联系了，我也可以为事件取其他任何名字。

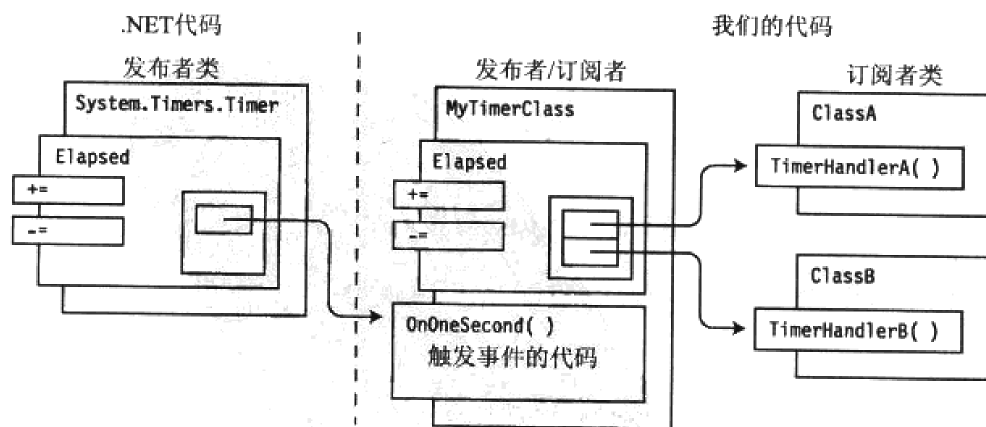



图16-5　MyTimer类的代码结构

Timer的属性之一是Interval，它是double类型的，并指定了触发事件间隔的毫秒数。代码用

到的另外一个属性是Enabled，它是bool类型的，用来开启和停止计时器。

实际代码如下。我之前唯一没有提到过的就是一个叫做MyPrivateTimer的私有计时器字段和类的构造函数。构造函数用于设置内置计时器并附加OnOneSecond事件处理程序。

```
public class MyTimerClass
{
   public event EventHandler Elapsed;

   private void OnOneSecond(object obj, EventArgs e)
   {
      if (Elapsed != null)
         Elapsed(obj, e);
   }

   //------------
   private System.Timers.Timer MyPrivateTimer;    // 私有计时器

   public MyTimerClass()                          // 构造函数
   {
      MyPrivateTimer = new System.Timers.Timer(); // 创建私有计时器

      // 下面的语句将上面的OnOnSecond设置成了
      // 类计时器的Elapsed事件的事件处理程序
      // 它与我们上面声明的Elapsed事件完全无关
      MyPrivateTimer.Elapsed += OnOneSecond;      // 附加事件处理程序

      MyPrivateTimer.Interval = 1000;

      MyPrivateTimer.Enabled = true;
   }
}
```

## 16.8 事件访问器

本章介绍的最后一个主题是事件访问器。之前我提到过，+=和-=运算符是事件允许的唯一运算符。看到这里我们应该知道，这些运算符有预定义的行为。

然而，我们可以修改这些运算符的行为，而且当使用它们时，可以让事件执行任何我们希望的自定义代码。我们可以通过为事件定义事件访问器来实现。

- 有两个访问器：add和remove。
- 声明事件的访问器看上去和声明一个属性差不多。

下面的示例演示了具有访问器的事件声明。两个访问器都有叫做value的隐式值参数，它接受实例或静态方法的引用。

```
public event EventHandler Elapsed
{
    add
    {
        ...                             //执行=+运算符的代码
    }
    remove
    {
        ...                             //执行-=运算符的代码
    }
}
```

声明了事件访问器之后，事件不包含任何内嵌委托对象。我们必须实现自己的机制来存储和移除事件注册的方法。

事件访问器表现为void方法，也就是不能使用会返回值的return语句。

# 第17章 接　口

**本章内容**

- 什么是接口
- 声明接口
- 实现接口
- 接口是引用类型
- 接口和as运算符
- 实现多个接口
- 实现具有重复成员的接口
- 多个接口的引用
- 派生成员作为实现
- 显式接口成员实现
- 接口可以继承接口

## 17.1 什么是接口

接口是表示一组函数成员而不实现成员的引用类型。其他类型——类和结构可以实现接口。

为了对接口有一些感性认识，让我们先来看一个已经定义好的接口。BCL声明了一个叫做IComparable的接口，它的声明如下面的代码所示。注意，接口主体中包含了一个CompareTo方法，它接受一个object类型的参数。尽管方法有名字、参数和返回类型，却没有实现。实现被一个分号来替代了。

```
         关键字      接口名
           ↓          ↓
public interface IComparable
{
   int CompareTo( object obj );
}                             ↑
                   分号代替了方法实现
```

图17-1所示为IComparable接口的结构。灰色部分的CompareTo方法表示它不包含实现。

接口声明没有为CompareTo方法提供实现，但IComparable接口的.NET文档描述了方法应该在创建实现接口的类或结构时被实现。也就是说，当方法CompareTo被调用时，应该返回如下值之一：

- 如果当前对象比参数对象小，则返回负值；
- 如果当前对象比参数对象大，则返回正值；
- 如果两个对象在比较中被认为是相等的，则返回0。

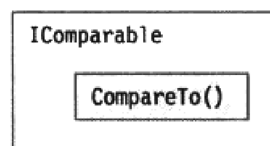


图17-1　Icomparable接口图示

## 使用 ICompatable 接口的示例

要理解接口的意义以及为什么它是有用的，让我们先来看看如下代码，它接受了一个没有排序的整数数组并且按升序进行排序。

- 第一行代码创建了5个无序整数数组。
- 第二行代码使用了Array类的静态Sort方法来排序元素。
- foreach循环输出它们，显示数字以升序排序。

```
var myInt = new [] { 20, 4, 16, 9, 2 };

Array.Sort(myInt);                          // 按大小排序

foreach (var i in myInt)                    // 输出它们
   Console.Write("{0} ", i);
```

这段代码产生了如下的输出：

```
2 4 9 16 20
```

Sort方法在int数组上工作得很好，但是如果我们尝试在自己的类上使用会发生什么呢？如下所示：

```
class MyClass
{
   public int TheValue;
}
   ...
MyClass[] mc = new MyClass[5];          // 创建一个有5个元素的数组
   ...                                  // 创建并初始化元素

Array.Sort(mc);                         // 尝试使用Sort时抛出异常
```

当运行这段代码时，它会抛出一个异常。为什么它对于int数组可行，而对于MyClass对象的数组不可行呢？

Sort不工作的原因是，对于用户自定义对象的数组而言，它不知道如何比较用户自定义对象和确定它们的次序。它需要数组中的对象实现IComparable接口。在Sort运行时，它通过调用元素的CompareTo方法并传入另外一个元素的引用作为参数来实现数组的一个元素与另一个元素的比较。

int类型实现IComparable，但是MyClass没有实现，所以当Sort尝试调用MyClass不存在的CompareTo方法时会产生异常。

我们可以通过让类实现IComparable来使Sort方法可以用于MyClass类型的对象。要实现接口、

类或结构，必须做两件事情：

- 必须在基类列表后面列出接口名称。
- 必须为接口提供每一个成员的实现。

例如，下面代码更新了MyClass来实现IComparable接口。注意下面关于代码的内容：

- 接口名称列在类声明的基类列表中。
- 类实现了一个名称为CompareTo的方法，它的参数类型和返回类型与这些接口成员一致。
- CompareTo方法的实现遵循接口说明的定义。也就是根据它的值与传入方法对象的值进行比较，返回-1、1或0。

```
            基类列表中的接口名
                  ↓
class MyClass : IComparable
{
   public int TheValue;

   public int CompareTo(object obj)   // 引用方法的实现
   {
      MyClass mc = (MyClass)obj;
      if (this.TheValue < mc.TheValue) return -1;
      if (this.TheValue > mc.TheValue) return  1;
      return 0;
   }
}
```

图17-2演示了更新后的类。从灰色接口方法到类方法的箭头表示接口方法不包含代码，实现在类级别的方法中。

注意，MyClass实现了IComparable接口，Sort可用于这个类。补充一下，如果仅仅声明CompareTo方法是不够的，必须实现接口，也就是把接口名称放在基类列表中。

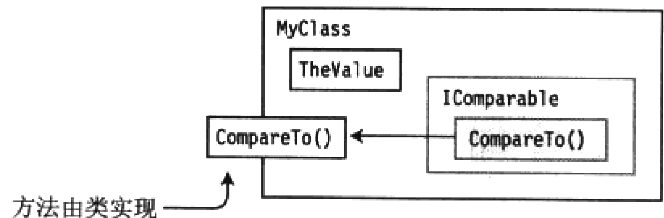



图17-2 在MyClass中实现IComparable

下面显示了更新后的完整代码，现在就可以使用Sort方法来排序MyClass对象数组了。Main创建并初始化了MyClass对象的数组并且把它们输出，然后调用了Sort并且重新输出，以显示它们已经被排序了。

```
class MyClass : IComparable                    // 类实现引用
{
   public int TheValue;
   public int CompareTo(object obj)            // 实现方法
   {
      MyClass mc = (MyClass)obj;
      if (this.TheValue < mc.TheValue) return -1;
      if (this.TheValue > mc.TheValue) return 1;
      return 0;
```

```
    }
}

class Program
{
   static void PrintOut(string s, MyClass[] mc)
   {
      Console.Write(s);
      foreach (var m in mc)
         Console.Write("{0} ", m.TheValue);
      Console.WriteLine("");
   }

   static void Main()
   {
      var myInt = new [] { 20, 4, 16, 9, 2 };

      MyClass[] mcArr = new MyClass[5];      // 创建MyClass对象的数组
      for (int i = 0; i < 5; i++)            // 初始化数组
      {
         mcArr[i] = new MyClass();
         mcArr[i].TheValue = myInt[i];
      }
      PrintOut("Initial Order:  ", mcArr);  // 输出初始数组
      Array.Sort(mcArr);                     // 数组排序
      PrintOut("Sorted Order:   ", mcArr);  // 输出排序后的数组
   }
}
```

这段代码产生了如下的输出：

```
Initial Order:  20 4 16 9 2
Sorted Order:   2 4 9 16 20
```

## 17.2　声明接口

前面部分的内容使用BCL中已经声明的接口。在这部分内容中，我们会来看看如何声明接口。关于接口，需要知道的重要事项如下所示。

- 接口声明不包含数据成员。
- 接口声明只能包含如下类型的静态成员函数的声明：
  - 方法
  - 属性
  - 事件
  - 索引
- 这些函数成员的声明不能包含任何实现代码，而在每一个成员声明的主体后必须使用分号。
- 按照惯例，接口名称必须从大写的I开始（比如ISaveable）。

和类以及结构一样，接口声明还可以分隔成分部接口声明，与第6章中提到的“分部类”一样。例如，下面的代码演示了具有两个方法成员接口的声明：

```
 关键字    接口名称
   ↓          ↓
interface IMyInterface1                    分号代替了主体
{                                               ↓
   int    DoStuff     ( int nVar1, long lVar2 );
   double DoOtherStuff( string s, long x );
}                                          ↑
                                      分号代替了主体
```

接口的访问性和接口成员的访问性之间有一些重要区别：

- 接口声明可以有任何的访问修饰符public、protected、internal或private。
- 然而，接口的成员是隐式public的，不允许有任何访问修饰符，包括public。

```
接口允许访问修饰符
  ↓
public interface IMyInterface2
{
   private int Method1( int nVar1, long lVar2 );          // 错误
}     ↑
 接口成员不允许访问修饰符
```

## 17.3 实现接口

只有类和结构才能实现接口。如Sort示例所示，要实现接口，类或结构必须：

- 在基类列表中包括接口名称。
- 为每一个接口的成员提供实现。

例如，如下代码演示了MyClass类的新声明，它实现了前面内容中声明的IMyInterface1接口。注意，接口名称列在在冒号后的基类列表中，并且类提供了接口成员的实际实现。

```
              冒号   接口名
               ↓        ↓
class MyClass: IMyInterface1
{
   int    DoStuff     ( int nVar1, long lVar2 )
   { ... }                                      // 实现代码

   double DoOtherStuff( string s, long x )
   { ... }                                      // 实现代码
}
```

关于实现接口，需要了解的重要事项如下：

- 如果类实现了接口，它必须实现接口的所有成员。

- 如果类从基类继承并实现接口，基类列表中的基类名称必须放在任何接口之前，如下所示。

```
                基类必须放在最前面          接口名
                       ↓                      ↓
class Derived : MyBaseClass, IIfc1, IEnumerable, IEnumerator
{
   ...
}
```

## 简单接口的示例

如下代码声明了一个叫IIfc1的接口，它包含了一个叫做PrintOut的成员。MyClass类通过把IIfc1接口列在它的基类列表中并提供一个与接口成员的签名和返回类型相匹配的PrintOut方法来实现它。Main创建了类对象并调用对象的方法。

```
interface IIfc1      分号代替了主体                    // 声明接口
{                          ↓
   void PrintOut(string s);
}
                实现接口
                   ↓
class MyClass : IIfc1                                  // 声明类
{
   public void PrintOut(string s)                      // 实现
   {
      Console.WriteLine("Calling through:  {0}", s);
   }
}

class Program
{
   static void Main()
   {
      MyClass mc = new MyClass();                       // 创建实例
      mc.PrintOut("object.");                           // 调用方法
   }
}
```

这段代码产生了如下的输出：

```
Calling through:  object.
```

# 17.4　接口是引用类型

接口不仅仅是类或结构要实现的成员列表。它是一个引用类型。

我们不能直接通过类对象的成员访问接口。然而，我们可以通过把类对象引用强制转换

为接口类型来获取指向接口的引用。一旦有了接口的引用，我们就可以使用点号来调用接口的方法。

例如，如下代码给出了一个从类对象引用获取接口引用的示例：

- 在第一个语句中，mc变量是一个实现了IIFc1接口的类对象的引用。语句强制转换那个引用为指向接口的引用，并将它赋值给变量ifc。
- 在第二个语句中，使用指向接口的引用来调用实现的方法。

```
 接口        转换为借口
  ↓             ↓
IIfc1 ifc = (IIfc1) mc;                  // 获取接口的引用
        ↑           ↑
     接口引用    类对象引用

ifc.PrintOut ("interface");              // 使用接口的引用调用方法
      ↑
 使用语法通过接口引用调用
```

例如，如下的代码声明了接口以及一个实现它的类。Main中的代码创建了类的对象，并通过类对象调用实现方法。它还创建了接口类型的变量，强制把类对象的引用转换成接口类型的引用，并通过接口的引用来调用实现方法。图17-3演示了类和接口的引用。

```
interface IIfc1
{
   void PrintOut(string s);
}

class MyClass: IIfc1
{
   public void PrintOut(string s)
   {
      Console.WriteLine("Calling through:  {0}", s);
   }
}

class Program
{
   static void Main()
   {
      MyClass mc = new MyClass();  // 创建类对象
      mc.PrintOut("object.");      // 调用类对象的实现方法

      IIfc1 ifc = (IIfc1)mc;       // 将类对象的引用转换为接口类型的引用
      ifc.PrintOut("interface.");  // 调用引用方法
   }
}
```

这段代码产生了如下的输出：

```
Calling through:  object.
Calling through:  interface.
```

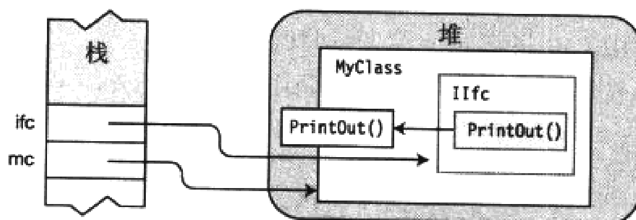



图17-3　类对象的引用以及接口的引用

## 17.5　接口和 as 运算符

在前面的内容中，我们已经知道了可以使用强制转换运算符来获取对象接口的引用，另一个更好的方式是使用as运算符。as运算符在18章中会详细介绍，但我在这里会提一下，因为这是与接口配合使用的好机会。

如果我们尝试强制转换类对象引用为类未实现的接口的引用，强制转换操作会抛出一个异常。我们可以通过使用as运算符来避免这个问题。具体方法如下所示。

- 如果类实现了接口，表达式返回指向接口的引用。
- 如果类没有实现接口，表达式返回null而不是抛出异常。

如下代码演示了as运算符的使用。第一行使用了as运算符来从类对象获取接口引用。表达式的结果会把b的值设置为null或ILiveBirth接口的引用。

第二行代码检测了b的值，如果它不是null，则执行命令来调用接口成员方法。

```
             类对象引用   接口名
                   ↓          ↓
ILiveBirth b = a as ILiveBirth;
           ↑     ↑
       接口引用   运算符

if (b != null)
   Console.WriteLine("Baby is called: {0}", b.BabyCalled());
```

## 17.6　实现多个接口

到现在为止，类只实现了单个引用。

- 类或结构可以实现任意数量的接口。
- 所有实现的接口必须列在基类列表中并以逗号分隔（如果有基类名称，则在其之后）。

例如，如下的代码演示了MyData类，它实现了两个接口：IDataStore和IDataRetrieve。图17-4演示了MyData类中多个接口的实现。

```
interface IDataRetrieve { int GetData(); }          //声明接口
interface IDataStore { void SetData( int x ); }     //声明接口
                         接口         接口
                          ↓           ↓
class MyData: IDataRetrieve, IDataStore             //声明类
{
   int Mem1;                                        //声明字段
   public int  GetData()        { return Mem1; }
   public void SetData( int x ) { Mem1 = x;    }
}

class Program
{
   static void Main()                               // Main
   {
      MyData data = new MyData();
      data.SetData( 5 );
      Console.WriteLine("Value = {0}", data.GetData());
   }
}
```

这段代码产生了如下的输出：

```
Value = 5
```

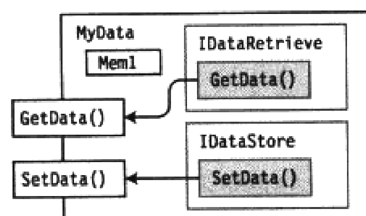



图17-4　类实现了多个接口

## 17.7　实现具有重复成员的接口

由于类可以实现任意数量的接口，有可能两个或多个接口成员都有相同的签名和返回类型。编译器如何处理这样的情况呢？

例如，假设我们有两个接口IIfc1和IIfc2。每一个接口都有一个名称为PrintOut的方法，具有相同的签名和返回类型。如果我们要创建实现两个接口的类，怎么样处理重复接口的方法呢？

```
interface IIfc1
{
   void PrintOut(string s);
}

interface IIfc2
{
```

```
    void PrintOut(string t);
}
```

答案是：如果一个类实现了多个接口，那么其中一些接口有相同签名和返回类型的成员。类可以实现单个成员来满足所有包含重复成员的接口。

例如，如下代码演示了MyClass类的声明，它实现了IIfc1和IIfc2。实现方法PrintOut2满足了两个接口的需求。

```
class MyClass : IIfc1, IIfc2               // 实现两个接口
{
   public void PrintOut(string s)          // 两个接口的单一实现
   {
      Console.WriteLine("Calling through:  {0}", s);
   }
}

class Program
{
   static void Main()
   {
      MyClass mc = new MyClass();
      mc.PrintOut("object.");
   }
}
```

这段代码产生了如下的输出：

```
Calling through:  object.
```

图17-5演示了利用单个类级别的方法的实现来实现重复接口的方法。

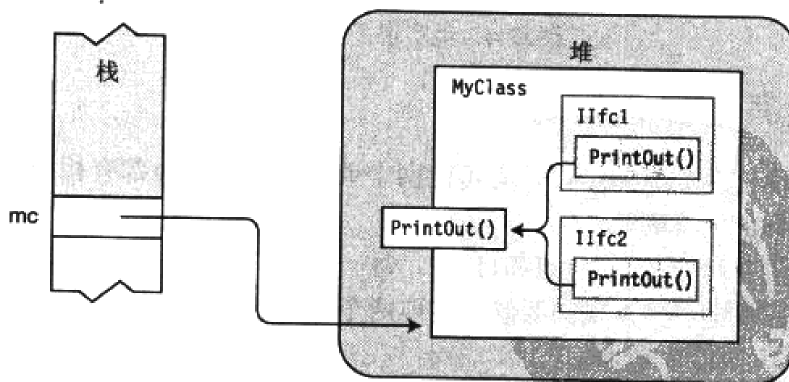



图17-5　由同一个类成员实现多个接口

## 17.8 多个接口的引用

我们已经在之前的内容中知道了接口是引用类型，并且可以通过强制转换对象引用为接口类

型的引用来获取一个指向接口的引用。如果类实现了多个接口，我们可以获取每一个的独立引用。

例如，下面的类是实现了两个具有单个PrintOut方法的接口。Main中的代码以三种方式调用了PrintOut。

- ❑ 通过类对象。
- ❑ 通过指向IIfc1接口的引用。
- ❑ 通过指向IIfc2接口的引用。

图17-6演示了类对象以及指向IIfc1和IIfc2的引用。

```
interface IIfc1 { void PrintOut(string s); }
interface IIfc2 { void PrintOut(string s); }

class MyClass : IIfc1, IIfc2 {
   public void PrintOut(string s)
   {
      Console.WriteLine("Calling through:  {0}", s);
   }
}

class Program {
   static void Main() {
      MyClass mc = new MyClass();
      IIfc1 ifc1 = (IIfc1) mc;                    // 获取IIfc1的引用
      IIfc2 ifc2 = (IIfc2) mc;                    // 获取IIfc2的引用

      mc.PrintOut("object.");                     // 从类对象调用
      ifc1.PrintOut("interface 1.");              // 从IIfc1调用
      ifc2.PrintOut("interface 2.");              // 从IIfc1调用
   }
}
```

这段代码产生了如下的输出：

```
Calling through:  object.
Calling through:  interface 1.
Calling through:  interface 2.
```

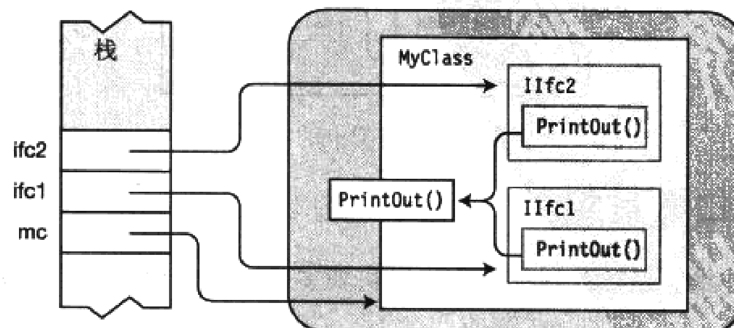



图17-6 分离类中不同接口的引用

## 17.9　派生成员作为实现

实现接口的类可以从它的基类继承实现的代码。例如，如下的代码演示了类从它的基类代码继承了实现。

- IIfc1是一个具有PrintOut方法成员的接口。
- MyBaseClass包含了一个叫做PrintOut的接口，它和IIfc1的方法相匹配。
- Derived类有一个空的声明主体，但它派生自MyBaseClass，并在基类列表中包含了IIfc1。
- 即使Derived的声明主体是空的，基类中的代码还是能满足实现接口方法的需求。

```
interface IIfc1 { void PrintOut(string s); }

class MyBaseClass
{
   public void PrintOut(string s)
   {
      Console.WriteLine("Calling through:  {0}", s);
   }
}

class Derived : MyBaseClass, IIfc1
{
}

class Program {
   static void Main()
   {
      Derived d = new Derived();
      d.PrintOut("object.");
   }
}
```

图17-7演示了前面的代码。注意，始自IIfc1的箭头指向了基类中的代码。

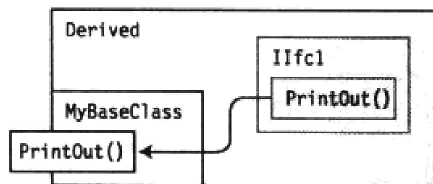



图17-7　基类中的实现

## 17.10　显式接口成员实现

在之前的内容中，我们已经看到单个类可以实现多个接口需要的所有成员，如图17-5和图17-6所示。

但是，如果我们希望为每一个接口分离实现该怎么做呢？在这种情况下，我们可以创建显式

接口成员实现。显式接口成员实现有如下特性：

- 与所有接口实现相似，它被类和结构用来实现接口。
- 它使用限定接口名称来声明，由接口名称和成员名称以及它们中间的点分隔符号构成。

如下代码显示了声明显式接口成员实现的语法。由MyClass实现的两个接口都实现了各自版本的PrintOut方法。

```
class MyClass : IIfc1, IIfc2
{         限定接口名
              ↓
   void IIfc1.PrintOut (string s)                // 显式实现
   { ... }

   void IIfc2.PrintOut (string s)                // 显式实现
   { ... }
}
```

图17-8演示了类和接口。注意，表示显式接口成员实现的方框不是灰色的，因为它们现在表示实际代码。

例如，如下代码中的MyClass为两个接口的成员声明了显式接口成员实现。注意，在这个示例中只有一个显式接口成员实现，没有类级别的实现。

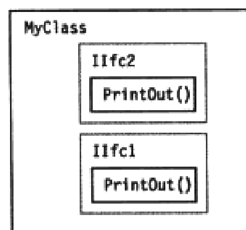



图17-8 显式接口成员实现

```
interface IIfc1 { void PrintOut(string s); }
interface IIfc2 { void PrintOut(string t); }

class MyClass : IIfc1, IIfc2
{         限定接口名
              ↓
   void IIfc1.PrintOut(string s)          // 显式接口成员实现
   {
      Console.WriteLine("IIfc1:  {0}", s);
   }
          限定接口名
              ↓
   void IIfc2.PrintOut(string s)          // 显式接口成员实现
   {
      Console.WriteLine("IIfc2:  {0}", s);
   }
```

```
    }

    class Program
    {
        static void Main()
        {
            MyClass mc = new MyClass();

            IIfc1 ifc1 = (IIfc1) mc;                 // 获取IIfc1的引用
            ifc1.PrintOut("interface 1.");           // 调用显式实现

            IIfc2 ifc2 = (IIfc2) mc;                 // 获取IIfc2的引用
            ifc2.PrintOut("interface 2.");           // 调用显式实现
        }
    }
```

这段代码产生了如下的输出：

```
IIfc1:  interface 1.
IIfc2:  interface 2.
```

图17-9演示了这段代码。注意，在图中接口方法没有指向类级别实现，而是包含了自己的代码。

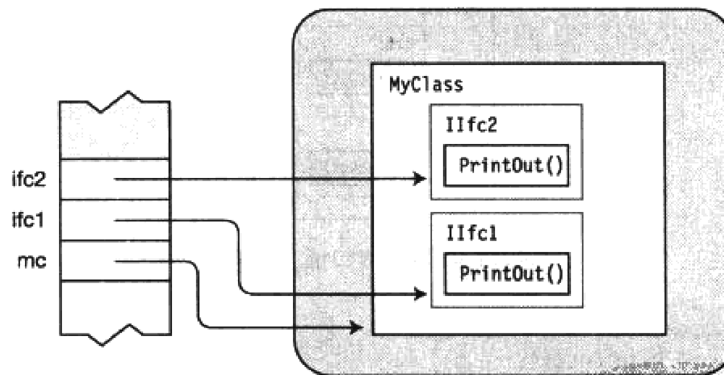



图17-9　具有显式接口成员实现的接口引用

如果有显式接口成员实现，类级别的实现是允许的，但不是必须的。显式实现满足了类或结构必须实现的方法。因此，我们可以有如下三种实现场景：

- 类级别实现。
- 显式接口成员实现。
- 类级别和显式接口成员实现。

## 访问显式接口成员实现

显式接口成员实现只可以通过指向接口的引用来访问。也就是说，其他的类成员都不可以直

接访问它们。

例如，如下代码演示了MyClass类的声明，它使用显式实现实现了IIfc1接口。注意，即使是MyClass的另一成员Method1，也不可以直接访问显式实现。

- Method1的前两行代码产生了编译错误，因为方法在尝试直接访问实现。
- 只有Method1的最后一行代码才可以编译，因此它强制转换当前对象的引用（this）为接口类型的引用，并使用这个指向接口的引用来调用显式接口实现。

```
class MyClass : IIfc1
{
   void IIfc1.PrintOut(string s)        // 显式接口实现
   {
      Console.WriteLine("IIfc1");
   }

   public void Method1()
   {
      PrintOut("...");                  // 编译错误
      this.PrintOut("...");             // 编译错误

      ((IIfc1)this).PrintOut("...");    // 调用方法
   }      ↑
}      转换为接口引用
```

这个限制对继承产生了重要问题。由于其他类不能直接访问显式接口成员实现，衍生类的成员也不能直接访问它们。它们必须总是通过接口的引用来访问。

## 17.11 接口可以继承接口

之前我们已经知道接口实现可以从基类被继承,而接口本身也可以从一个或多个接口继承。

- 要指定某个接口继承其他的接口，应在接口声明中把基接口以逗号分隔的列表形式放在接口名称的冒号之后，如下所示。

```
                  冒号        接口列表
                   ↓             ↓
interface IDataIO : IDataRetrieve, IDataStore
{ ...
```

- 与类不同，它只在基类列表中只能有一个类名，而接口可以在基接口列表中有任意多个接口。
  - 列表中的接口本身可以有被继承的接口。
  - 结果接口包含它声明的所有接口和所有基接口的成员。

图17-10的代码演示了三个接口的声明。IDataIO接口从前两个继承。图右边部分显示IDataIO包含了另外两个接口。

```
interface IDataRetrieve
{ int GetData( ); }

interface IDataStore
{ void SetData( int x ); }

// 从前两个接口继承
interface IDataIO: IDataRetrieve, IDataStore
{
}

class MyData: IDataIO {
   int nPrivateData;
   public int GetData( )
         { return nPrivateData; }
   public void SetData( int x )
         { nPrivateData = x; }
}

class Program {
   static void Main( ) {
      MyData data = new MyData ();
      data.SetData( 5 );
      Console.WriteLine("{0}", data.GetData());
   }
}
```

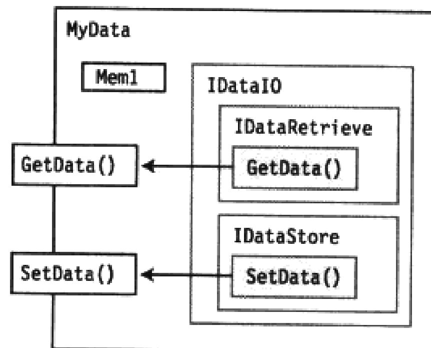



图17-10 类实现的接口继承了多个接口

## 不同类实现一个接口的示例

如下代码演示了已经介绍过的接口的一些方面。程序声明一个名称为Animal的类，它被作为其他一些类的基类来表示各种类型的动物。它还声明了一个叫做ILiveBirth的接口。

Cat、Dog和Bird类都从Animal基类继承。Cat和Dog都实现了ILiveBirth接口，而Bird类没有。

在Main中，程序创建了Animal对象的数组并对三个动物类的对象进行填充。最后，程序遍历数组并使用as运算符获取指向ILiveBirth接口的引用，并调用了BabyCalled方法。

```
interface ILiveBirth
{
   string BabyCalled();
}

class Animal { }

class Cat : Animal, ILiveBirth
{
   string ILiveBirth.BabyCalled()
   { return "kitten"; }
}

class Dog : Animal, ILiveBirth
{
   string ILiveBirth.BabyCalled()
   { return "puppy"; }
```

```
    }

    class Bird : Animal
    {
    }

    class Program
    {
       static void Main()
       {
          Animal[] animalArray = new Animal[3];    // 创建Animal数组
          animalArray[0] = new Cat();              // 插入Cat类对象
          animalArray[1] = new Bird();             // 插入Bird类对象
          animalArray[2] = new Dog();              // 插入Dog类对象
          foreach( Animal a in animalArray )
          {
             ILiveBirth b = a as ILiveBirth;
             if (b != null)
                Console.WriteLine("Baby is called: {0}", b.BabyCalled());
          }
       }
    }
```

这段代码产生了如下的输出：

```
Baby is called: kitten
Baby is called: puppy
```

图17-11演示了内存中的数组和对象。

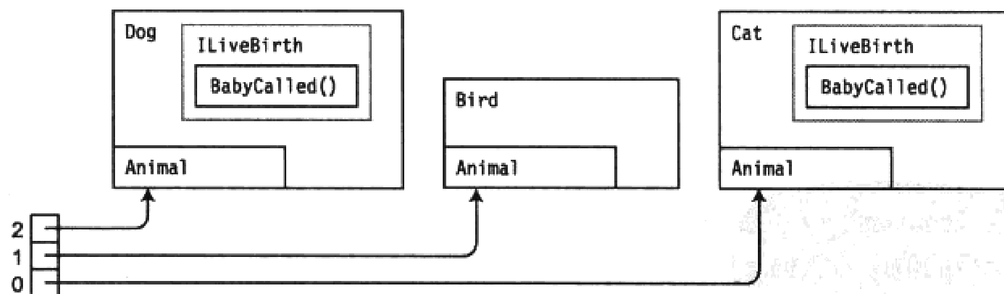



图17-11 Animal基类的不同对象类型在数组中的布局

# 第18章 转换



**本章内容**

- 什么是转换
- 隐式转换
- 显式转换和强制转换
- 转换的类型
- 数字的转换
- 引用转换
- 装箱转换
- 拆箱转换
- 用户自定义转换
- is运算符
- as运算符

## 18.1 什么是转换

要理解什么是转换，让我们先从声明两个不同类型的变量，然后把一个变量（源）的值赋值给另外一个变量（目标）的简单示例开始讲起。在赋值之前，源的值必须转换成目标类型的值。图18-1演示了类型转换。

- 转换（conversion）是接受一个类型的值并使用它作为另一个类型的等价值的过程。
- 转换后的值应和源值一样的，但它是目标类型。

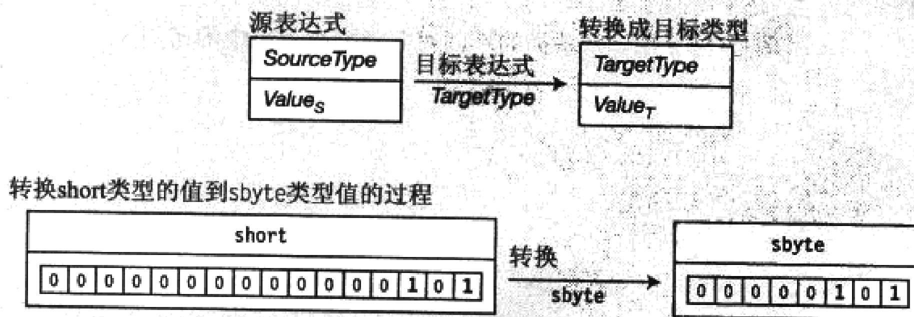



图18-1　类型转换

例如，图18-2中的代码给出了两个不同类型的变量的声明。

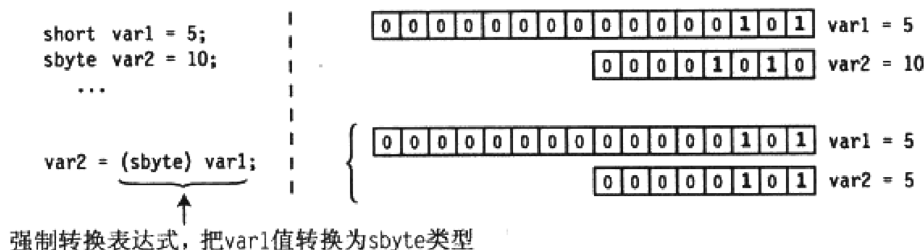



图18-2　从short转换为sbyte

- var1是short类型的，16位有符号整数，初始值为5。var2是sbyte类型的，8位有符号整数，初始值为10。
- 第三行代码把var1赋值给var2。由于它们是两种不同的类型，在进行赋值之前，var1的值类型必须先转换为与var2的值类型相同。这将通过强制转换表达式来实现，稍后我们就会看到。
- 注意，var1的类型和值都没有改变。尽管它被称之为转换，但只是代表源值作为目标类型来使用，不是源值转换为目标类型。

## 18.2　隐式转换

有些类型的转换不会丢失数据或精度。例如，在不丢失数据的情况下，很容易把8位的值转换为16位的值。

- 语言会自动做这些转换，这叫做隐式转换。
- 从位数更少的源转换为位数更多的目标类型时，目标中多出来的位需要用0或1填充。
- 当从更小的无符号类型转换为更大的无符号类型时，目标类型多出来的最高位都以0进行填充，这叫做零扩展。

图18-3演示了使用零扩展把8位的10轻化为16位的10。

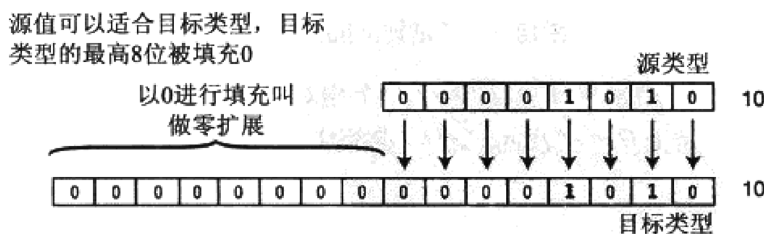



图18-3　无符号转换中的零扩展

对于有符号类型的转换而言，额外的高位用源表达式的符号位进行填充。

- 这样就维持了被转换的值的正确符号和大小。
- 这被叫做符号扩展，如图18-4所演示的，第一个是10，后面一个是-10。

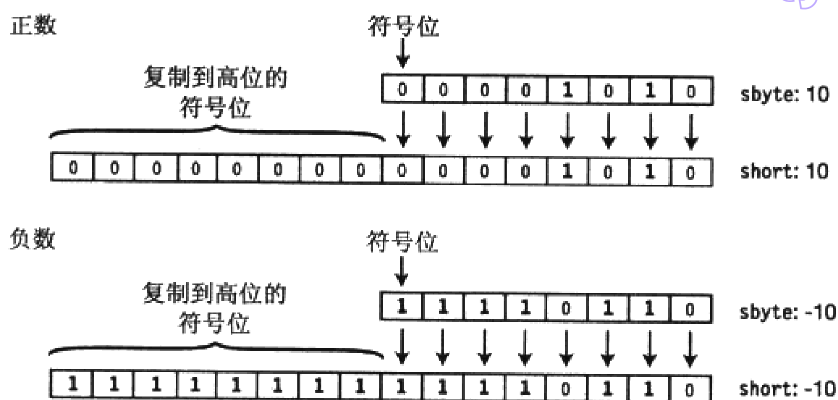



图18-4　有符号转换中的符号扩展

## 18.3　显式转换和强制转换

如果要把短类型转换为长类型，对于长类型来说，保存所有短类型的字符很简单。然而，在其他情况下，目标类型也许不能在不损失数据的情况下提供源值。

例如，假设我们希望把ushort值转化为byte。

- ushort可以保存任何0～65 535之间的值。
- byte只能保存0～255之间的值。
- 只要希望转换的ushort值小于255，那么就不会损失数据。然而，如果更大，最高位的数据会丢失。

例如，图18-5演示了尝试把值为1 365的ushort类型转换为byte类型会导致数据丢失。

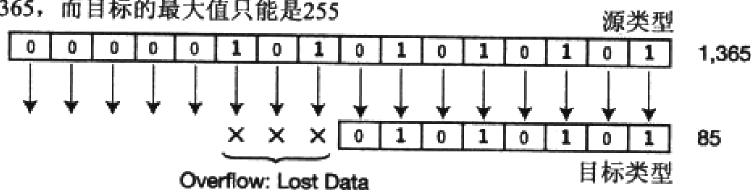



图18-5　尝试把ushort转化为byte

很明显，只有当无符号16位ushort的值是一个相对小一些的数字（40%）时，才能在不损失数据的情况下被安全转换为无符号8位byte类型。数据中的其他结果会溢出（overflow），产生其他值。

### 强制转换

对于预定义的类型，C#会自动将一个数据类型转换为另一个数据类型，但只是针对那些从源类型到目标类型不会发生数据丢失的情况。也就是说，对于源类型的任意值在被转换成目标类型时会丢失值的情况，语言是不会提供这两种类型的自动转换的。如果希望对这样的类型进行转换，就必须使用显式转换。这叫做强制转换表达式。

如下代码给出了一个强制转换表达式的示例。如果它把var1的值转换为sbyte类型。强制转

换表达式的构成如下所示。

- ❑ 包含目标类型的一组配对圆括号。
- ❑ 圆括号后是源表达式。

```
目标类型
   ↓
( sbyte ) var1;
            ↑
         源表达式
```

如果我们使用强制转换表达式，就意味着我们要承担执行操作可能会引起的丢失数据的责任。从本质上是说："不管是否会发生数据丢失，我知道在做什么，总之进行转换吧。"（然而，需要确信你知道在做什么。）

例如，图18-6演示了强制转换表达式将两个ushort类型的值转换为byte类型。对于第一种情况，没有数据丢失。对于第二种情况，最高位丢失了，得到的值是85，很明显不等于源值1 365。

```
ushort sh = 10;
byte sb = (byte) sh;
Console.WriteLine
     ("sb:  {0} = 0x{0:X}", sb);

sh = 1365;
sb = (byte) sh;
Console.WriteLine
     ("sb:  {0} = 0x{0:X}", sb);
```

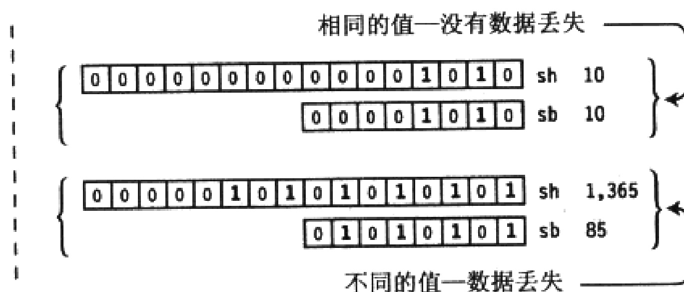



图18-6 强制转换ushort为byte

图中代码的输出如下：

```
sb:  10 = 0xA
sb:  85 = 0x55
```

## 18.4 转换的类型

有很多标准的、预定义的用于数字和引用类型的转换。图18-7演示了这些不同的转换类型。

- ❑ 除了标准转换，还可以为自定义类型定义隐式转换和显式转换。
- ❑ 还有一个预定义的转换类型，叫做装箱，它转换任何值类型到：
  - ■ object类型
  - ■ System.ValueType类型
- ❑ 拆箱转换装箱的值到原始类型。

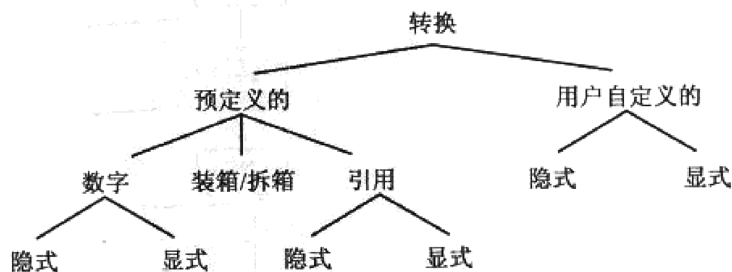



图18-7 转换的类型

## 18.5 数字的转换

任何数字类型都可以转换为其他数字类型，如图18-8所示。一些转换是隐式的，而另外一些转换则应该是显式的。

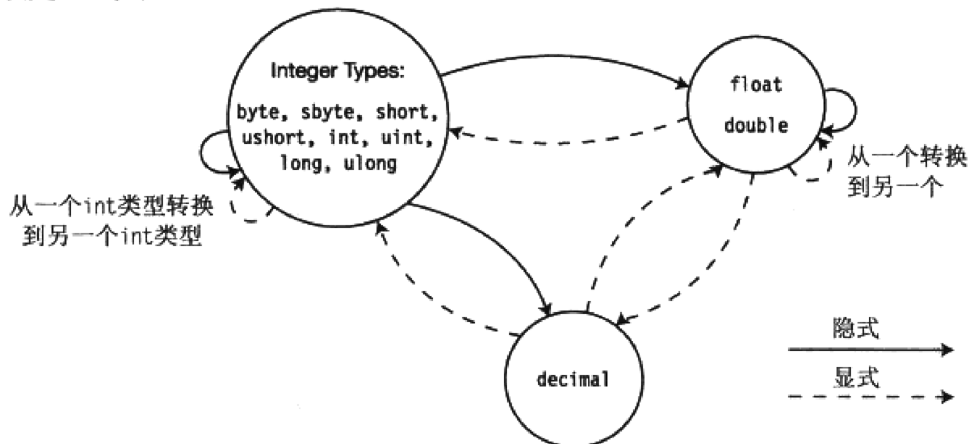



图18-8 数字转换

### 18.5.1 隐式数字转换

图18-9演示了隐式数字转换。

- 如果有路径，从源类型到目标类型可以按照箭头进行隐式转换。
- 任何在从源类型到目标类型的箭头方向上没有路径的数字转换可能都是显式转换。

图中所演示的，正如我们期望的那样，占据较少位的数字类型可以隐式转换为占据较多位的数字类型。

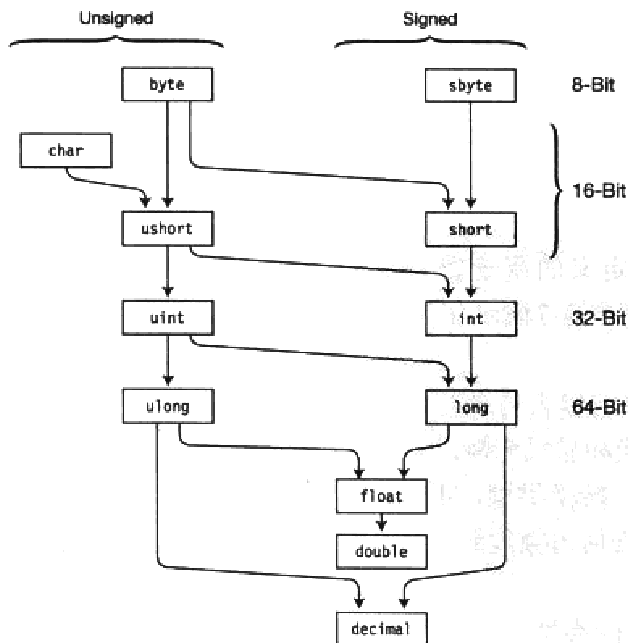



图18-9 隐式数字转换

## 18.5.2 溢出检测上下文

我们已经知道了，显式转换可能会丢失数据并且不能在目标类型中同等地表示源值。对于整数类型，C#给我们提供了选择运行时是否应该在进行类型转换时检测结果溢出的能力。这将通过checked运算符和checked语句来实现。

- 代码片段是否被检查称作溢出检测上下文。
  - 如果我们指定一个表达式或一段代码为checked，CLR会在转换产生溢出时抛出一个OverflowException异常。
  - 如果代码不是checked，转换会继续而不管是否产生溢出。
- 默认的溢出检测上下文不是checked。

### 1. checked和unchecked运算符

checked和unchecked运算符控制表达式的溢出检测上下文。表达式放置在一对圆括号内并且不能是一个方法。语法如下所示：

```
checked   （表达式）
unchecked （表达式）
```

例如，如下代码执行了相同的转换——第一个在checked运算符内，而第二个在unchecked运算符内。

- 在unchecked上下文中，会忽略溢出，结果值是208。
- 在checked上下文中，抛出了OverflowException异常。

```
ushort sh = 2000;
byte   sb;

sb = unchecked ( (byte) sh );          //大多数重要的位丢失了
Console.WriteLine("sb: {0}", sb);

sb =   checked ( (byte) sh );          //抛出OverflowException异常
Console.WriteLine("sb: {0}", sb);
```

这段代码产生了如下的输出：

```
sb: 208

Unhandled Exception: System.OverflowException: Arithmetic operation resulted➥
    in an overflow. at Test1.Test.Main() in C:\Programs\Test1\Program.cs:line 21
```

### 2. Checked语句和unchecked语句

checked和unchecked运算符用于圆括号内的单个表达式。checked和unchecked语句执行相同的功能，但控制一块代码的所有转换，而不是单个表达式。

checked语句和unchecked语句可以被嵌套在任意层次。

例如，如下代码使用了checked语句和unchecked语句，并产生了与之前使用checked和unchecked表达式的示例相同的结果。然而，在这种情况下，影响的是一段代码，而不仅仅是一个表达式。

```
byte   sb;
ushort sh = 2000;

unchecked                                      // 设置unchecked
{
   sb = (byte) sh;
   Console.WriteLine("sb: {0}", sb);

   checked                                     // 设置checked
   {
      sb = (byte) sh;
      Console.WriteLine("sb: {0}", sh);
   }
}
```

### 18.5.3　显式数字转换

我们已经知道了，隐式转换之所以能自动从源表达式转换到目标类型是因为不可能丢失数据。然而，对于显式转换而言，就可能丢失数据。因此，作为一个程序员，知道发生数据丢失时转换会如何处理很重要。

在本节中，我们会来看看各种显式数字转换。图18-10演示了图18-8中显式转换的子集。

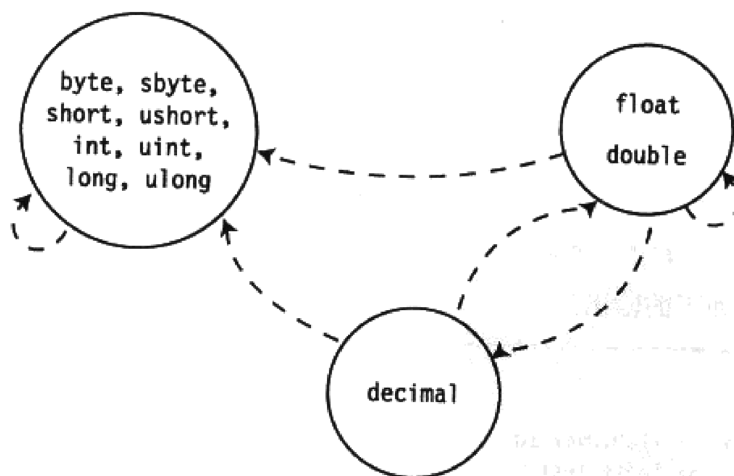



图18-10　显式数字转换

#### 1. 整数到整数

图18-10演示了整数到整数的显式转换的行为。在checked的情况下，如果转换会丢失数据，操作会抛出一个OverflowException异常。在unchecked的情况下，丢失的位不会发出警告。

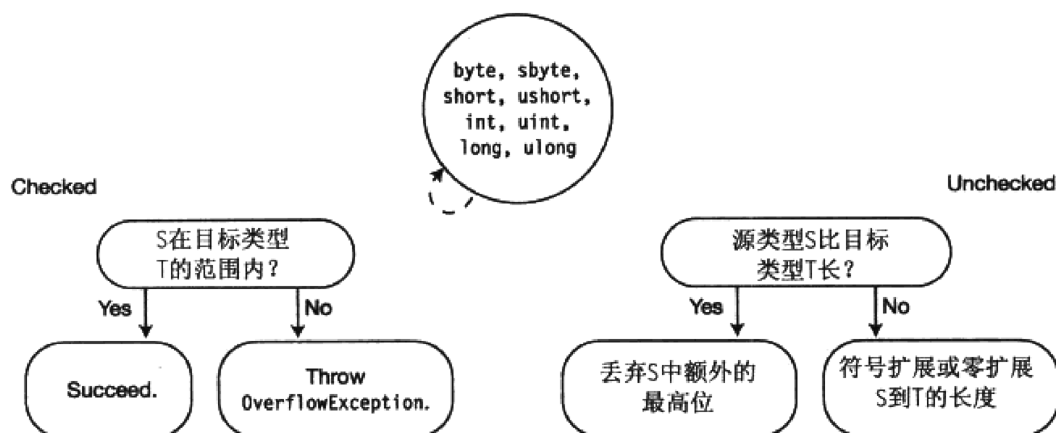



图18-11 整数到整数的显式转换

### 2. float或double转到整数

当把浮点类型转换为整数类型时，值会向零的方向取整也就是截断为最接近的整数。图18-12演示了转换条件。如果截断后的值不在目标类型的范围内：

- 如果溢出检测上下文是checked，则CLR会抛出OverflowException异常。
- 如果上下文是unchecked，则C#将不定义它的值应该是什么。

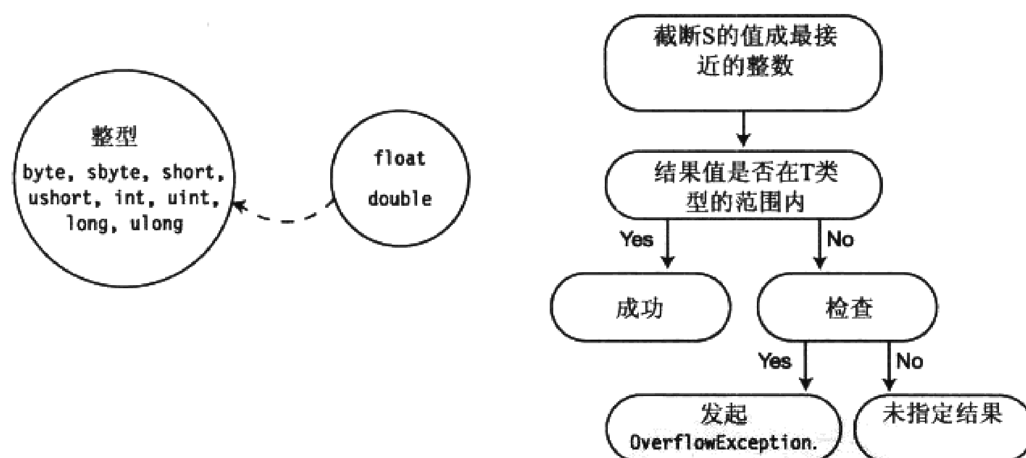



图18-12 转换float或double为整数类型

### 3. decimal到整数

当从decimal转换到整数类型时，如果结果值不在目标类型的范围内，则CLR会抛出Overflow-Exception。图18-13演示了转化条件。

### 4. double到float

float类型的值占32位，而double类型的值占64位。double类型的值被舍入到最接近的float类型的值。图18-14演示了转换条件。

- 如果值太小而不能用float表示，那么值会被设置为正或负0。

❑ 如果值太大而不能用float表示，那么值会被设置为正或负无穷大。

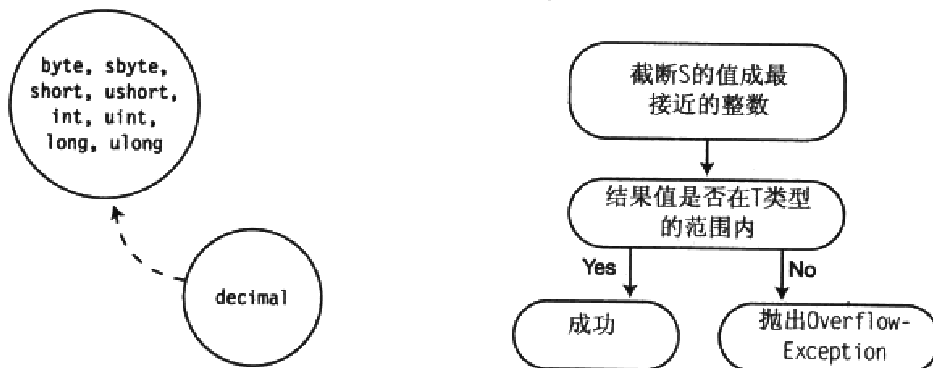



图18-13 转换decimal到整数

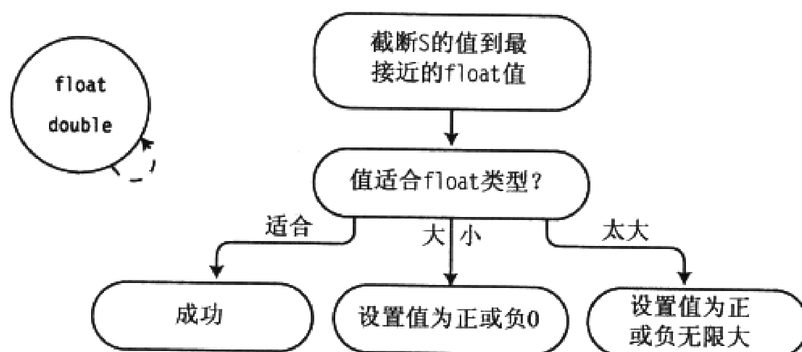



图18-14 转换double到float

### 5. float或double到decimal

图18-15演示了转换float类型到decimal类型的转换条件。

❑ 如果值太小而不能用decimal类型表示，那么值会被设置为0。

❑ 如果值太大，那么CLR会抛出OverflowException异常。

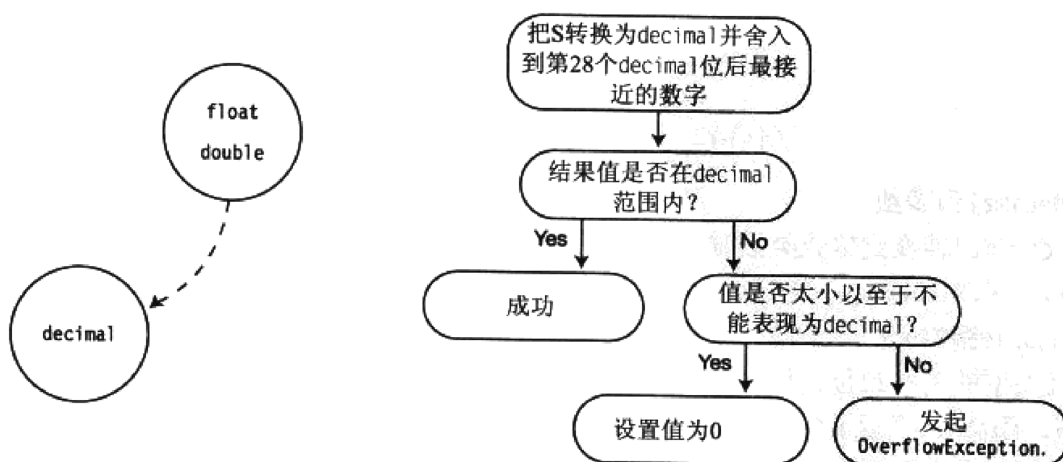



图18-15 转换float或double到decimal

### 6. decimal到float或double

从decimal类型转换到float类型总是会成功。然而，可能会损失精度。图18-16演示了转换条件。

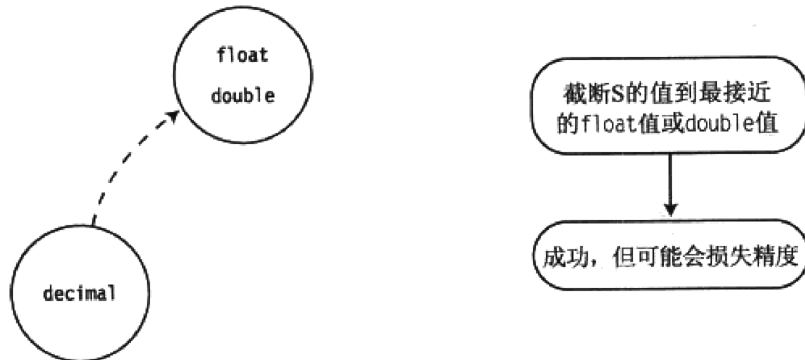



图18-16 转换decimal到float或double

## 18.6 引用转换

至此，我们已经知道引用类型对象由内存中的两部分组成：引用和数据。

- 由引用保存的那部分信息是它指向的数据类型。
- 引用转换接受源引用并返回一个指向堆中同一位置的引用，但是把引用“标记”为其他类型。

例如，如下代码给出了两个引用变量：myVar1和myVar2，它们指向内存中的相同对象。代码如图18-17所示。

- 对于myVar1，它引用的对象看上去是B类型的对象——其实就是。
- 对于myVar2，同样的对象看上去像类型A的对象。
  - 即使它实际指向B类型的对象，它也不能看到B扩展A的部分，因此不能看到Field2。
  - 第二个WriteLine语句因此会产生编译错误。

注意，“转换”不会改变myVar1。

```
class A    { public int Field1; }

class B: A { public int Field2; }

class Program
{
   static void Main( )
   {
      B myVar1 = new B();
     作为A类的引用返回myVar1的引用
                  ↓
      A myVar2 = (A) myVar1;

      Console.WriteLine("{0}", myVar2.Field1);          // 正确
```

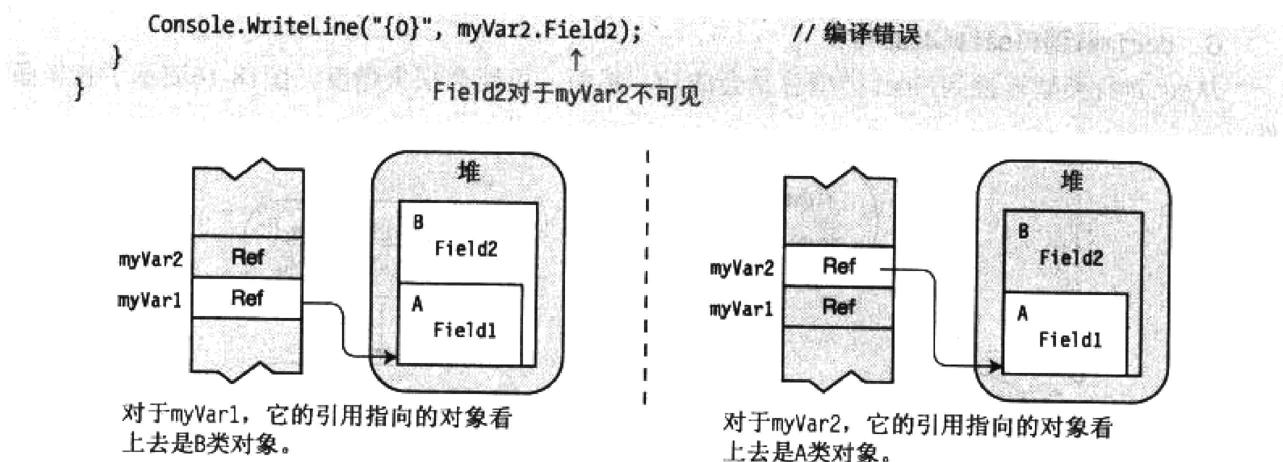



图18-17　引用转换返回与对象关联的不同类型

## 18.6.1　隐式引用转换

就好像语言为我们自动实现的隐式数字转换一样，还有隐式引用转换，如图18-18所示。

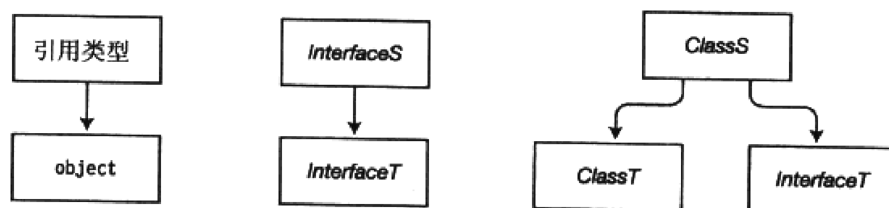



图18-18　类和接口的隐式转换

- 所有引用类型可以被隐式转换为object类型。
- 任何类型可以隐式转换到它继承的接口。
- 类可以隐式转换到：
  - 它继承的链中的任何类。
  - 它实现的任何接口。

委托可以隐式转换成图18-19所示的.NET BCL类和接口。

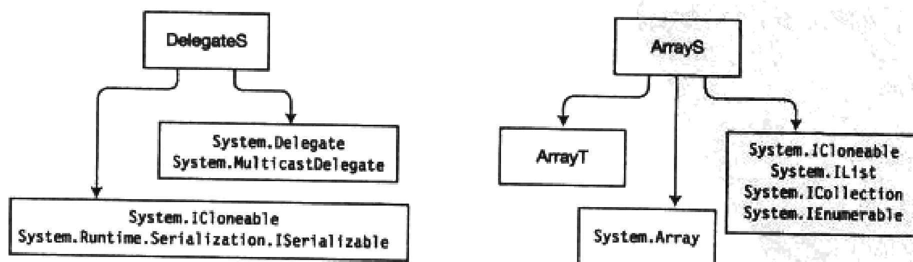



图18-19　委托和数组的隐式转换

ArrayS数组，其中的元素是Ts类型，可以隐式转换成：

❑ 图18-19所示的.NET BCL类和接口。

❑ 另一个数组ArrayT，其中的元素是Tt类型（如果下面的命题都成立）。

  ■ 两个数组都有一样的维度。

  ■ 元素类型Ts和Tt都是引用类型，不是值类型。

  ■ 在类型Ts和Tt中存在隐式转换。

### 18.6.2 显式引用转换

显式引用转换是从一个普通类型到一个更精确类型的引用转换。

❑ 显式转换包括：

  ■ 从object到任何引用类型的转换。

  ■ 从基类到从它继承的类的转换。

❑ 倒转图18-18和图18-19的箭头方向，可以演示显式引用转换。

如果转换的类型不受限制，很可能会导致我们尝试引用在内存中实际并不存在的类成员。然而，编译器确实允许这样的转换。到那时，如果系统在运行时遇到它们则会抛出一个异常。

例如，图18-20中的代码将基类A的引用转换到它的衍生类B，并且把它赋值给变量myVar2。

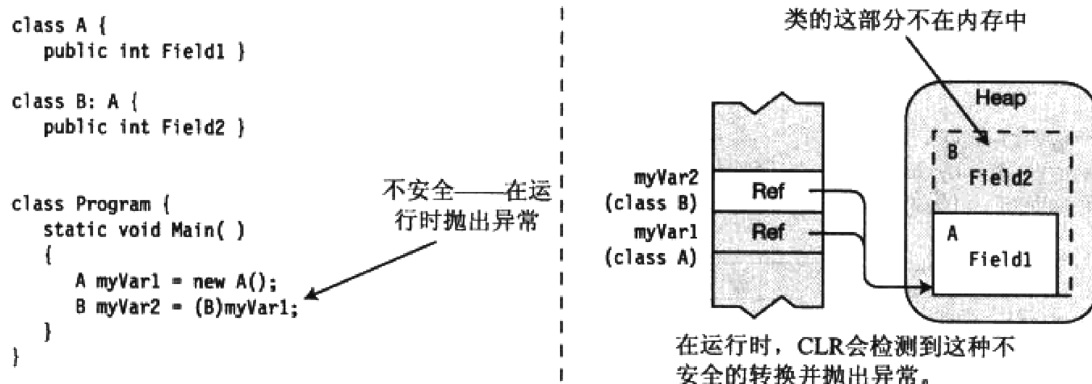



图18-20 无效的转换抛出运行时异常

❑ 如果myVar2尝试访问Field2，它会尝试访问对象中“B部分”的字段（它不在内存中），这会导致内存错误。

❑ 运行时会捕获到这种不正确的强制转换并且抛出InvalidCastException异常。然而，注意，它不会导致编译错误。

### 18.6.3 有效显式引用转换

在运行时能成功进行（也就是不抛出InvalidCastException异常）的显式转换有三种情况。

第一种情况：显式转换是没有必要的——也就是说，语言已经为我们进行了隐式转换。例如，在下面的代码中，显式转换是没有必要的，因为从衍生类到基类的转换总是隐式转换的。

```
class A { }
class B: A { }
   ...
B myVar1 = new B();
A myVar2 = (A) myVar1;      //不必转换，因为A是B的基类
```

第二种情况：源引用是null。例如，在下面的代码中，即使转换基类的引用到衍生类的引用通常会是不安全的，但是由于源引用是null，这种转换还是允许的。

```
class A { }
class B: A { }
   ...
A myVar1 = null;
B myVar2 = (B) myVar1;      // 允许转换，因为myVar1为空
```

第三种情况：由源引用指向的实际数据可以被安全地进行隐式转换。如下代码给出了这个示例，图18-21演示了这段代码。

- 第二行中的隐式转换使myVar2看上去像指向A类型的数据，其实它指向的是B类型的数据对象。
- 第三行中的显式转换把基类引用强制转换为它的衍生类的引用。通常，这会产生异常。然而，在这种情况下，被指向的对象实际就是B类型的数据项。

```
B myVar1 = new B();
A myVar2 = myVar1;      // 将myVar1隐式转换为A类型
B myVar3 = (B)myVar2;  // 该转换是允许的，因为数据是B类型的
```

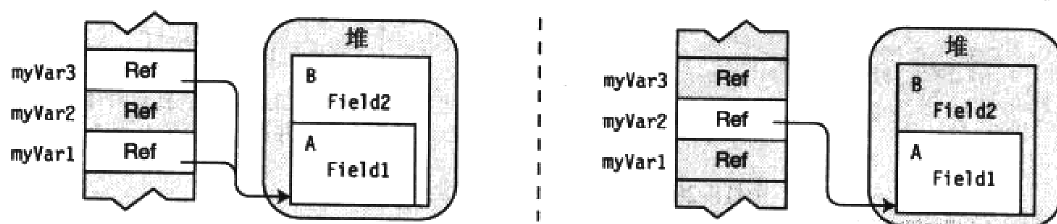



图18-21　强制转换到安全类型

## 18.7　装箱转换

包括值类型在内的所有C#类型都派生自object类型。然而，值类型是高效轻量的类型，因为默认情况下在堆上不包括它们的对象组件。然而，如果需要对象组件，我们可以使用装箱（boxing）。装箱是一种接受值类型的值，根据这个值在堆上创建一个完整的引用类型对象并返回对象引用的隐式转换。

例如，图18-22演示了三行代码。

- 前两行代码声明并初始化了值类型变量i和引用类型变量oi。
- 在代码的第三行，我们希望把变量i的值赋给oi。但是oi是引用类型的变量，我们必须在

堆上分配一个对象的引用。然而，变量i是值类型，不需要在堆上有对象的引用。

- 因此，系统将i的值装箱如下：
  - 在堆上创建了int类型的对象；
  - 将i的值复制到int对象；
  - 返回int对象的引用，让oi作为引用保存。

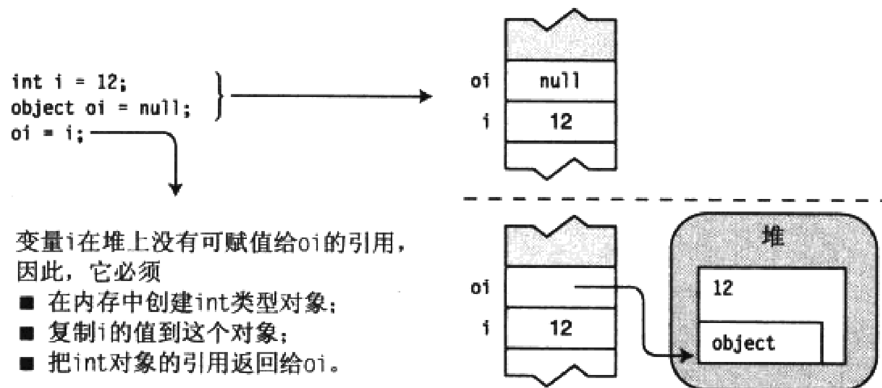



图18-22 装箱从值类型创建了完整的引用类型

## 装箱创建一份副本

一个有关装箱的普遍误解是在被装箱的项上发生了一些操作。其实不是，它返回了副本的值的引用类型。在装箱产生之后，有两份值——原始值类型和副本的引用类型，每一个都可以独立操作。

如下代码演示了独立操作每一个副本的值。图18-23演示了这段代码。

- 第一行定义了值类型变量i并且初始化它的值为10。
- 第二行创建了引用类型变量oi，并且使用装箱后变量i的副本进行初始化。
- 代码的最后三行演示了i和oi是如何被独立操作的。

```
int i = 10;                    //创建并初始化值类型
  装箱i并把引用赋值给oi
          ↓
object oi = i;                 //创建并初始化引用类型
Console.WriteLine("i: {0}, io: {1}", i, oi);

i  = 12;
oi = 15;
Console.WriteLine("i: {0}, io: {1}", i, oi);
```

这段代码产生了如下的输出：

```
i: 10, io: 10
i: 12, io: 15
```

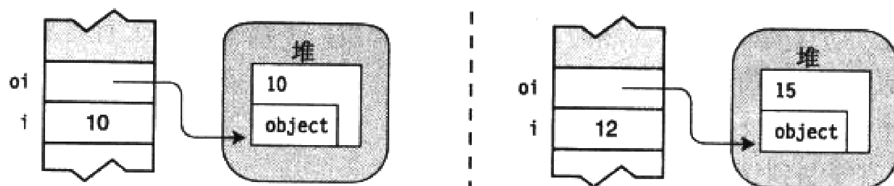


图18-23 装箱创建了一份可以被独立操作的副本

### 1. 装箱转换

图18-24演示了装箱转换。任何值类型ValueTypeS都可以被隐式转换为object类型、System.ValueType或InterfaceT（如果ValueTypeS实现了InterfaceT）。

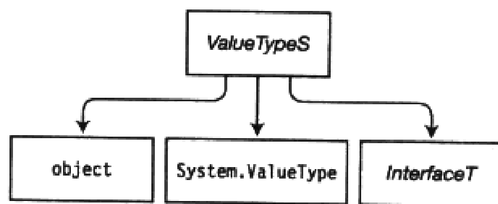


图18-24 装箱是值类型到引用类型的隐式转换

## 18.8 拆箱转换

拆箱（unboxing）是把装箱后的对象转换回值类型的过程。

- 拆箱是显式转换。
- 系统在把值拆箱成ValueTypeT时执行了如下的步骤：

(1) 它检测到要拆箱的对象实际是ValueTypeT的装箱值。

(2) 它把对象的值复制到变量。

例如，如下代码给出了拆箱一个值的示例。

- 值类型变量i被装箱并且赋值给引用类型变量oi。
- 变量oi然后被拆箱，它的值被赋值给值类型变量j。

```
static void Main()
{
    int i = 10;
      装箱i并把引用赋值给oi
              ↓
    object oi = i;
        拆箱oi并把值赋值给j
              ↓
    int j = (int) oi;
    Console.WriteLine("i: {0},  oi: {1},  j: {2}", i,  oi, j);
}
```

这段代码产生了如下的输出：

```
i: 10,  oi: 10,  j: 10
```

尝试拆箱一个值为非原始类型时会抛出一个InvalidCastException异常。

### 拆箱转换

图18-25显示了拆箱转换。

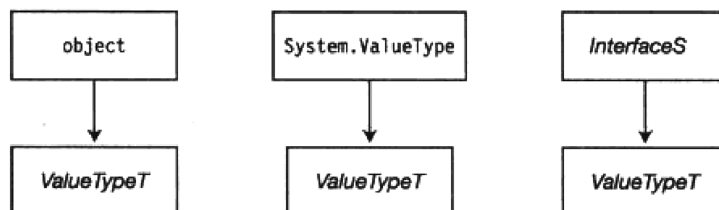



图18-25 拆箱转换

## 18.9 用户自定义转换

除了标准转换，我们还可以为类和结构定义隐式和显式转换。

用户自定义转换的语法如下：

- 除了implicit和explicit关键词之外，隐式和显示转换的声明语法是一样的。
- 需要public和static修饰符。

```
      必需的                 运算符    目标类型              源
        ↓                      ↓         ↓                    ↓
public static implicit operator TargetType ( SourceType Identifier )
{                  ↑
                隐式或显式
    ...
  return ObjectOfTargetType;
}
```

例如，下面的代码给出了一个转换语法的示例，它转换一个Person类型的对象为int。

```
public static implicit operator int(Person p)
{
  return p.Age;
}
```

### 18.9.1 用户自定义转换的约束

用户自定义转换有一些很重要的约束，最重要的如下所示。

- 只可以为类或结构定义用户自定义转换。
- 不能重定义标准隐式转换或显式转换。
- 对于源类型S和目标类型T，如下是命题是为真的：
  - S和T必须是不同类型。

- S和T不能通过继承关联。也就是说，S不能继承自T，而T也不能从S继承。
- S和T都不能是接口类型或object类型。
- 转换运算符必须是S或T的成员。

- 对于相同的源和目标类型，我们不能声明两个转换，一个是隐式转换而另外一个是显式转换。

### 18.9.2　用户自定义转换的示例

如下的代码定义了一个叫做Person的类，它包含了人的名字和年龄。这个类还定义了两个隐式转换，第一个将Person对象转换为int值，目标int值是人的年龄。第二个将int转换为Person对象。

```
class Person
{
   public string Name;
   public int    Age;
   public Person(string name, int age)
   {
      Name = name;
      Age = age;
   }

   public static implicit operator int(Person p)   // 将person转换为int
   {
      return p.Age;
   }

   public static implicit operator Person(int i)   // 将int转换为person
   {
      return new Person("Nemo", i);
   }
}

class Program
{
   static void Main( )
   {
      Person bill = new Person( "bill", 25);

      把Person对象转换为int
              ↓
      int age = bill;
      Console.WriteLine("Person Info: {0}, {1}", bill.Name, age);

        把int转换为Person对象
                ↓
      Person anon = 35;
```

```
        Console.WriteLine("Person Info: {0}, {1}", anon.Name, anon.Age);
    }
}
```

这段代码产生了如下的输出：

```
Person Info: bill, 25
Person Info: Nemo, 35
```

如果定义了相同的explicit运算符而不是implicit，我们需要使用强制转换表达式来进行转换，如下所示：

```
                显式
      ...         ↓
public static explicit operator int( Person p )
{
   return p.Age;
}

...

static void Main( )
{
      ... 必须强制转换表达式
                 ↓
   int age = (int) bill;
      ...
```

## 18.9.3 计算用户自定义转换

到目前为止讨论的用户自定义转换都是在单步内直接把源类型转换为目标类型对象，如图18-26所示。

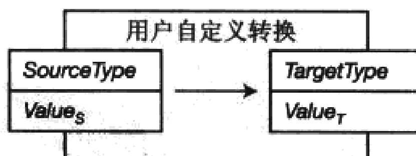



图18-26 单步用户自定义转换

但是，用户自定义转换也可以在完整转换中最多有3个步骤。图18-27演示了这3个步骤，它们包括：

- ❑ 预备标准转换
- ❑ 用户自定义转换
- ❑ 后续标准转换

在这个链中不可能有一个以上的用户自定义转换。

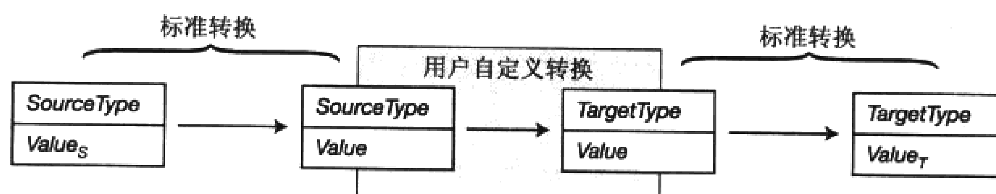



图18-27　多步用户自定义转换

## 18.9.4　多步用户自定义转换的示例

如下代码声明了Employee类，它继承自Person类。

- 之前内容中的代码示例已经声明了一个从Person类到int的用户自定义转换。如果从Employee到Person以及从int到float有标准转换，我们就可以从Employee转换到float。
  - 由于Employee是继承自Person的，从Employee到Person有标准转换。
  - 从int到float是隐式数字转换，也是标准转换。
- 由于链中的三部分都存在，我们就可以从Employee转换到flaot。图18-28演示了编译器如何进行转换。

```
class Employee : Person { }

class Person
{
   public string Name;
   public int    Age;


   public static implicit operator int(Person p)
   {
      return p.Age;
   }
}

class Program
{
   static void Main( )
   {
      Employee bill = new Employee();
      bill.Name = "William"; bill.Age = 25;
         把Employee转换为float
                 ↓
      float fVar = bill;

      Console.WriteLine("Person Info: {0}, {1}", bill.Name, fVar);
   }
}
```

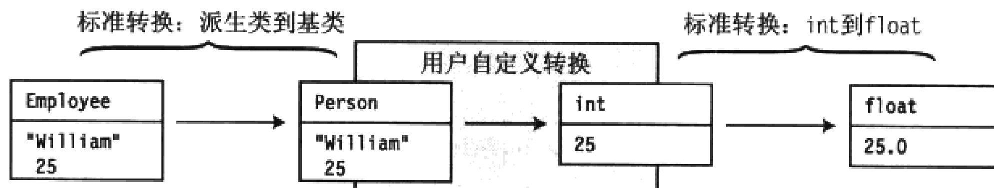



图18-28 从Employee转换到float

## 18.10 is 运算符

之前已经说过了，有些转换的请求是不成功的，并且会在运行时抛出一个InvalidCast-Exception异常。我们可以使用is运算符来检查转换是否会成功完成，从而避免盲目尝试转换。

is运算符的语法如下，Expr是源表达式：

```
    返回bool
       ↓
Expr is TargetType
```

如果Expr可以通过以下方式被成功转换为目标类型，运算符返回true：

- 引用转换
- 装箱转换
- 拆箱转换

例如，在如下代码中，使用is运算符来检测Employee类型的变量bill是否能转换为Person类型，然后进行合适的操作。

```
class Employee : Person { }
class Person
{
   public string Name = "Anonymous";
   public int Age     = 25;
}

class Program
{
   static void Main()
   {
      Employee bill = new Employee();
      Person p;

      // Check if variable bill can be converted to type Person
      if( bill is Person )
      {
         p = bill;
         Console.WriteLine("Person Info: {0}, {1}", p.Name, p.Age);
      }
   }
}
```

is运算符只可以用于引用转换以及装箱和拆箱转换，不能用于用户自定义转换。

## 18.11　as 运算符

as运算符和强制转换运算符类似，只是它不抛出异常。如果转换失败，它把目标引用设置为null而不是抛出异常。

as运算符的语法如下，其中：

- Expr是源表达式。
- TargetType是目标类型，它必须是引用类型。

```
       返回引用
          ↓
Expr as TargetType
```

由于as运算符返回引用表达式，它可以用作赋值中的源。

例如，我们使用as把Employee类型的变量bill转换为Person类型，并且赋值给一个Person类型的变量P。在使用它之前应该检查p是否为null。

```
class Employee : Person { }

class Person
{
   public string Name = "Anonymous";
   public int Age     = 25;
}

class Program
{
   static void Main()
   {
      Employee bill = new Employee();
      Person p;

      p = bill as Person;
      if( p != null )
      {
         Console.WriteLine("Person Info: {0}, {1}", p.Name, p.Age);
      }
   }
}
```

as运算符只能用于引用转换和装箱转换。它不能用于用户自定义转换或到值类型的转换。

# 第19章 泛 型

**本章内容**

- 什么是泛型
- C#中的泛型
- 泛型类
- 声明泛型类
- 创建构造类型
- 创建变量和实例
- 类型参数的约束
- 泛型结构
- 泛型接口
- 泛型委托
- 泛型方法
- 扩展方法和泛型类

## 19.1 什么是泛型

使用我们已经学习的语言结构，我们已经可以建立各种类型的强大对象。大部分情况下是声明一个类，然后封装需要的行为，最后创建这些类的实例。

到现在为止，所有在类声明中用到的类型都是特定的类型——或许是程序员定义的，或许是语言或BCL定义的。然而，很多时候，如果我们可以把类的行为提取或重构出来，使之不仅能应用到它们编码的数据类型上，而且还能应用到其他类型上的话，类就会更有用。

有了泛型我们就可以做到这一点了。我们可以重构代码并且额外增加一个抽象层，对于这样的代码来说，数据类型就不用硬编码了。这是专门为多段代码在不同的数据类型上执行相同指令的情况专门设计的。

也许这听起来比较抽象，让我们从一个示例开始，这样更清晰。

### 一个栈的示例

假设我们首先创建了如下的代码，它声明了一个叫做MyIntStack的类，该类实现了一个int类型的栈。它允许我们把int压入栈中，以及把它们弹出。

```
class MyIntStack
{
   int    StackPointer = 0;
   int[] StackArray;
    ↑
   整型              整型
                     ↓
   public void Push( int x )           // 输入类型：int
   {
      ...
   }      整型
          ↓
   public int Pop()                    // 返回类型：int
   {
      ...
   }
     ...
}
```

假设现在需要float类型的值有相同的功能，可以有几种方式来实现，一种方式是按照下面的步骤产生连续的代码：

- 复制并粘贴MyIntStack类的代码。
- 把类名改为MyFloatStack。
- 把整个类声明中相应的int声明改为float声明。

```
class MyFloatStack
{
   int    StackPointer = 0;
   float [] StackArray;
    ↑
   浮点型             浮点型
                      ↓
   public void Push( float x )         // 输入类型：float
   {
      ...
   }
         浮点型
          ↓
   public float Pop()                  // 返回类型：float
   {
      ...
   }

   ...
}
```

这个方法当然可行、但是很容易出错而且有如下缺点：

- 我们需要仔细检查类的每一个部分来看哪些类型的声明需要修改，哪些类型的声明需要保留。

- ❑ 每次需要新类型（long、double、string等）的栈类时，我们都需要重复这个过程。
- ❑ 在这些过程后，我们有了很多几乎具有相同代码的副本，占据了额外的空间。
- ❑ 调试和维护这些并行的实现不但复杂而且容易出错。

## 19.2 C#中的泛型

在C#2.0中，微软引入了泛型（generic）特性，它提供了一种更准确地使用有一种以上的类型的代码的方式。泛型允许我们声明类型参数化的代码，我们可以用不同的类型进行实例化。也就是说，我们可以用“类型占位符”来写代码，然后在创建类的实例时提供真实的类型。

至此，我们应该很清楚类型不是对象而是对象的模板这个概念了。同样地，泛型类型也不是类型，而是类型的模板。图19-1演示了这点。

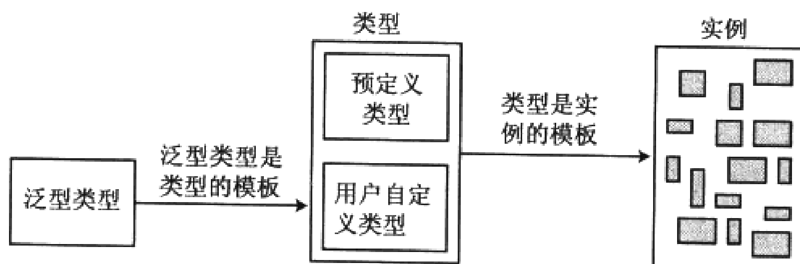



图19-1 泛型类型是类型的模板

C# 提供了5种泛型：类、结构、接口、委托和方法。注意，前面4个是类型，而方法是成员。

图19-2演示了泛型类型如何适合其他类型。

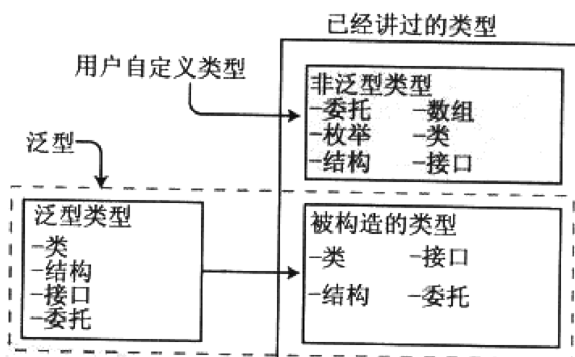



图19-2 泛型和用户自定义类型

### 继续栈的示例

在栈的示例中，MyIntStack和MyFloatStack两个类的主体声明都差不多，只不过在处理由栈保存的类型的地方有点不同。

- ❑ 在MyIntStack中，这些位置使用int类型占据。
- ❑ 在MyFloatStack中，这些位置被float占据。

我们从MyIntStack创建一个泛型类，通过如下步骤：

- ❑ 修改MyIntStack类的定义，把int替换为占位符T。
- ❑ 修改类名称为MyStack。
- ❑ 在类名后放置string<T>。

结果就是如下的泛型类声明。字符串由尖括号和T构成，T代表类型的占位符。（也不一定要是字母T，可以是任何标识符。）在类声明的主体中，每一个T都会被编译器替换为实际类型。

```
class MyStack <T>
{
```

```
    int StackPointer = 0;
    T [] StackArray;
    ↑
                    ↓
    public void Push(T x ) {...}

              ↓
    public T Pop() {...}
       ...
}
```

## 19.3　泛型类

既然我们已经见过了泛型类，让我们再来详细了解一下泛型类，并看看如何创建和使用它。

创建和使用常规的、非泛型的类的过程中有两个步骤：声明类并创建类的实例。但是泛型类不是实际的类，而是类的模板，所以我们必须先从它们构建实际的类类型，然后创建这个构建后的类类型的实例。

图19-3从一个较高的层面上演示了这个过程。如果你还不能完全清楚，不要紧，我们在之后的内容中会介绍每一部分。

- 在某些类型上使用占位符来声明一个类。
- 为占位符提供真实类型。这样就有了真实类型的定义，填补了所有的“空缺”。
- 从“填空后”的类定义创建实例。

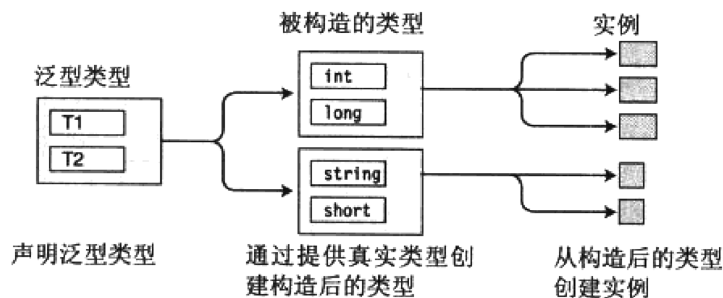



图19-3　从泛型类型创建实例

## 19.4　声明泛型类

声明一个简单的泛型类和声明普通类差不多，有如下的区别：

- 在类名之后放置一组尖括号。
- 在尖括号中用逗号分隔的占位符字符串来表示希望提供的类型。这被叫做类型参数（type parameter）。
- 在泛型类声明的主体中使用类型参数来表示应该被替代的类型。

例如，如下代码声明了一个叫做SomeClass的泛型类。类型参数被列在尖括号中，然后被当作真实类型在声明的主体中使用。

```
              类型参数
                 ↓
class SomeClass < T1, T2 >
{           通常在这些位置使用类型
             ↓                    ↓
   public T1 SomeVar  = new T1();
   public T2 OtherVar = new T2();
}            ↑                    ↑
            通常在这些位置使用类型
```

在泛型类型声明中没有特殊的关键词或标志。尖括号中的类型参数列表的表示将泛型类声明与普通类声明区分开。

## 19.5 创建构造类型

我们不能直接从泛型类型创建类对象。首先，我们需要告诉编译器使用哪些真实类型来替代占位符（类型参数）。编译器获取这些真实类型并从它创建一个真实类型对象。

要从泛型类构建类类型，列出类名字并在尖括号中提供真实类型来替代类型参数。要替代类型参数的真实类型叫做*类型实参*（type argument）。

```
           类型参数
              ↓
SomeClass< short, int >
```

编译器接受了类型实参并且替换泛型类主体中的相应类型参数，产生了构造类型——从它创建真实类型的实例。

图19-4左边演示了SomeClass泛型类型的声明。右边演示了使用类型实参short和int来创建构造类。

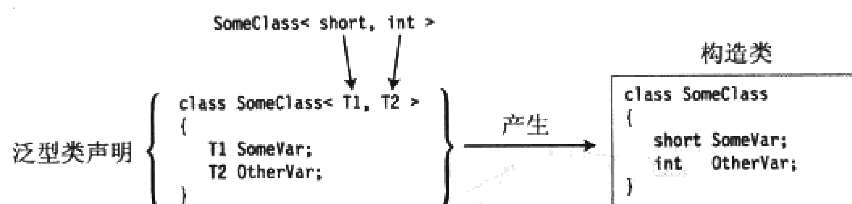



图19-4 为泛型类的所有类型参数提供类型实参来产生一个可以用来创建真实类对象的构造类

图19-5演示了类型参数和类型实参的区别。

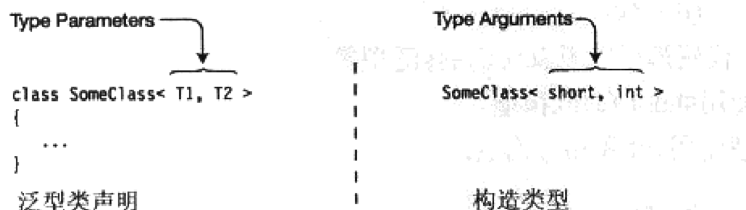



图19-5 类型参数与类型实参

- 泛型类声明有类型参数。
- 在创建构造类型时提供的真实类型是类型实参。

## 19.6　创建变量和实例

在创建引用和实例时构造类类型的使用和常规类型差不多。

例如，如下代码演示了两个类对象的创建。

- 第一行显示了普通非泛型类型对象的创建。这应该是我们非常熟悉的形式。
- 第二行代码显示了SomeClass泛型类型对象的创建，使用short和int类型进行实例化。这种形式和上面一行差不多，只不过把普通类型名改为构造类形式。
- 第三行和第二行的语法一样，没有在等号两边都列出构造类型，而是使用var关键词使编译器使用类型引用。

```
MyNonGenClass          myNGC = new MyNonGenClass        ();
      构造类                           构造类
        ↓                                ↓
SomeClass<short, int> mySc1 = new SomeClass<short, int>();
var                   mySc2 = new SomeClass<short, int>();
```

和非泛型类一样，引用和实例可以分开创建，如图19-6 所示。从图中还可以看出，内存中出现的情况与非泛型类是一样的。

- 泛型类声明下面的第一行在栈上为myInst分配了一个引用，值是null。
- 第二行在堆上分配实例，并且把引用赋值给变量。

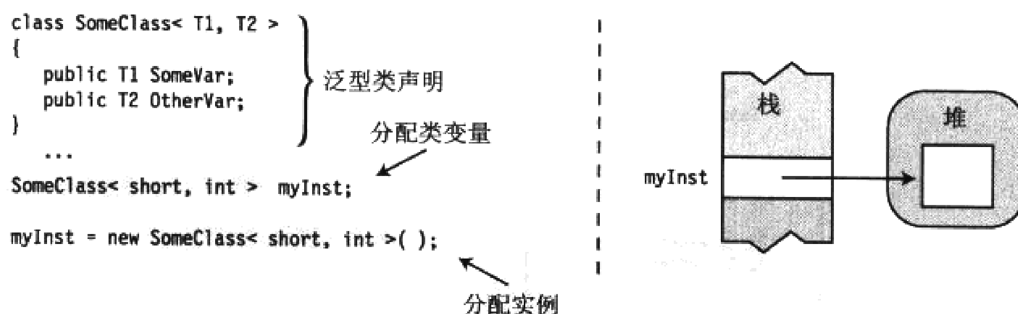



图19-6　使用构造类型来创建引用和实例

可以从同一个泛型类型构建出很多不同的类类型。每一个都有独立的类类型，就好像它们都有独立的非泛型类声明一样。

例如，下面的代码演示了从SomeClass泛型类创建两个类型。图19-7演示了代码。

- 一个类型使用short和int构造。
- 另一个类型使用int和long构造。

```
class SomeClass< T1, T2 >                        // 泛型类
{
```

```
    ...
}

class Program
{
   static void Main()
   {
      var first  =  new SomeClass<short, int >();    // 构造的类型
      var second =  new SomeClass<int,   long>();    // 构造的类型

         ...
```

```
class SomeClass< T1, T2 >
{
   T1 SomeVar;
   T2 OtherVar;
}
   ...
var first = new SomeClass<short, int> ( );

var second = new SomeClass<int, long> ( );
   ...
```

```
class SomeClass <short,int>
{
   short SomeVar;
   int   OtherVar;
}
```

```
class SomeClass <int,long>
{
   int   SomeVar;
   long  OtherVar;
}
```

图19-7　从泛型类创建的两个构造类

### 19.6.1　使用泛型的栈的示例

如下代码给出了一个使用泛型来实现栈的示例。Main方法定义了两个变量，stackInt和stack-String。使用int和string作为类型实参来创建这两个构造类型。

```
class MyStack<T>
{
   T[] StackArray;
   int StackPointer = 0;

   public void Push(T x) {
      if ( !IsStackFull )
         StackArray[StackPointer++] = x;
   }

   public T Pop() {
      return ( !IsStackEmpty )
         ? StackArray[--StackPointer]
         : StackArray[0];
   }

   const int MaxStack = 10;
   bool IsStackFull  { get{ return StackPointer >= MaxStack; } }
   bool IsStackEmpty { get{ return StackPointer <= 0; } }
```

```
    public MyStack() {
       StackArray = new T[MaxStack];
    }

    public void Print() {
       for (int i = StackPointer -1; i >= 0 ; i--)
          Console.WriteLine("   Value: {0}", StackArray[i]);
    }
 }

 class Program {
    static void Main() {
        var    stackInt = new MyStack<int>();
        var stackString = new MyStack<string>();

        stackInt.Push(3); stackInt.Push(5); stackInt.Push(7);
        stackInt.Print();
        stackString.Push("Generics are great!");
        stackString.Push("Hi there!  ");
        stackString.Print();
     }
 }
```

这段代码产生了如下的输出：

```
Value: 7
Value: 5
Value: 3
Value: Hi there!
Value: Generics are great!
```

## 19.6.2　比较泛型和非泛型栈

表19-1总结了原始非泛型版本的栈与最终泛型版本的栈之间的区别。图19-8演示了其中的一些区别。

表19-1　非泛型栈和泛型栈之间的区别

| | 非 泛 型 | 泛　型 |
|---|---|---|
| 源代码大小 | 更大：我们需要为每一种类型进行一个新的实现 | 更小：不管构造类型的数量有多少，我们只需要一个实现 |
| 可执行大小 | 无论每一个版本的栈是否会被使用，都会在编译的版本中出现 | 可执行文件中只会出现有构造类型的类型 |
| 写的难易度 | 易于书写 | 比较难写 |
| 维护的难易度 | 更容易出问题，因为所有修改需要应用到每一个可用的类型上 | 易于维护，因为只需要修改一个地方 |

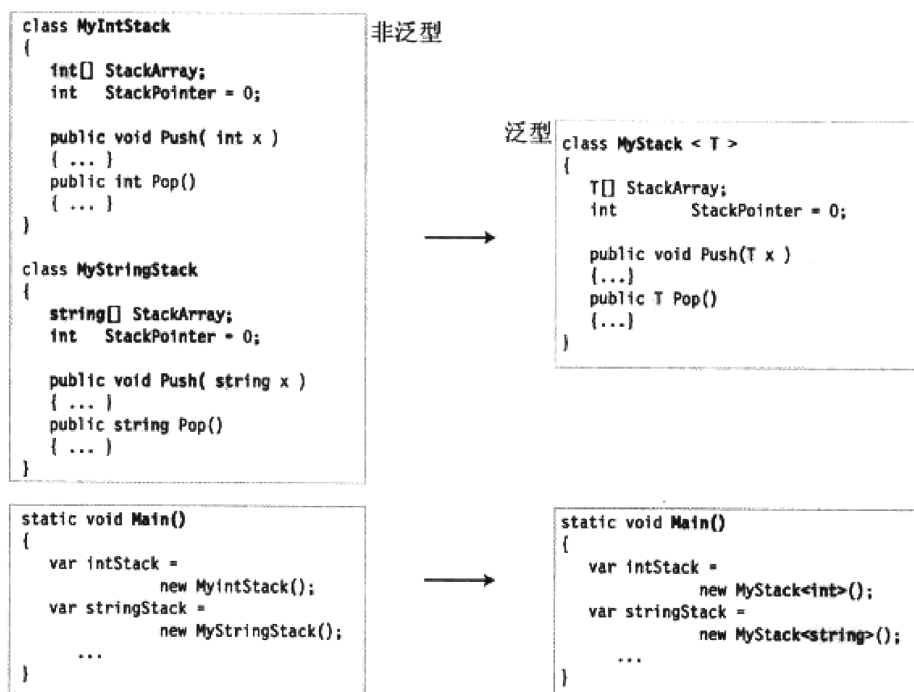



图19-8 非泛型栈和泛型栈

## 19.7 类型参数的约束

在泛型栈的示例中，栈除了保存和弹出它们包含的一些项之外没有做任何事情。它不会尝试添加、比较或做其他任何需要用到项本的身运算符的事情。理由还是很简单的，由于泛型栈不会知道它们保存的项的类型是什么，它不会知道这些类型实现的成员。

然而，所有的C#对象最终都从object类继承，因此，栈可以确认的是，这些保存的项都实现了object类的成员。它们包括ToString、Equals以及GetType。除了这些，它不知道还有哪些成员可用。

只要我们的代码不访问它处理的一些类型的对象（或者只要它始终是object类型的成员），泛型类就可以处理任何类型。符合约束的类型参数叫做**未绑定的类型参数**（unbounded type parameter）。然而，如果代码尝试使用其他成员，编译器会产生一个错误消息。

例如，如下代码声明了一个叫做Simple的类，它有一个叫做LessThan的方法，接受了两个泛型类型的变量。LessThan尝试使用小于运算符返回结果。但是由于不是所有的类都实现了小于运算符，所以编译器会产生一个错误消息。

```
class Simple<T>
{
   static public bool LessThan(T i1, T i2)
   {
      return i1 < i2;                    // 错误
   }
   ...
}
```

要让泛型变得更有用，我们需要提供额外的信息让编译器知道参数可以接受哪些类型。这些额外的信息叫做约束（constrain）。只有符合约束的实参才能用于类型参数。

## 19.7.1 Where子句

约束使用where子句列出。

- 每一个有约束的类型参数有自己的where子句。
- 如果形参有多个约束，它们在where子句中使用逗号分隔。

where子句的语法如下：

```
        类型参数              约束列表
           ↓                     ↓
where TypeParam : constraint, constraint, ...
                ↑
               冒号
```

有关where子句的要点如下：

- 它们在类型参数列表的关闭尖括号之后列出。
- 它们不使用逗号或其他符号分隔。
- 它们可以以任何次序列出。
- where是上下文关键词，所以可以在其他上下文中使用。

例如，如下泛型类有三个类型参数。T1是未绑定的，只有Customer类型或从Customer继承的类型的类才能用作T2的实参，而对于T3，只有实现IComparable接口的类才能用于类型实参。

```
                 未绑定  具有约束
                   ↓       ↓
class MyClass < T1, T2, T3 >       没有分隔符
                                     ↓
                where T2: Customer                  // T2的约束
                where T3: IComparable               // T3的约束
{                                      ↑
   ...                              没有分隔符
}
```

## 19.7.2 约束类型和次序

共有5种类型的约束，如表19-2所示。

表19-2　约束类型

| 约束类型 | 描　述 |
|---|---|
| 类名 | 只有这个类型的类或从它继承的类才能用作类型实参 |
| class | 任何引用类型，包括类、数组、委托和接口都可以用作实参 |
| struct | 任何值类型都可以被用作类型实参 |
| Interfacename | 只有这个接口或实现这个接口的类型才能用作实参 |
| new() | 任何带有无参公共构造函数的类型都可以用作实参。这叫做构造函数约束 |

where子句可以以任何次序列出。然而，where子句中的约束必须有特定的顺序，如图19-9所示：

- 最多只能有一个主约束，如果有则必须放在第一位。
- 可以有任意多的InterfaceName约束。
- 如果存在构造函数约束，则必须放在最后。

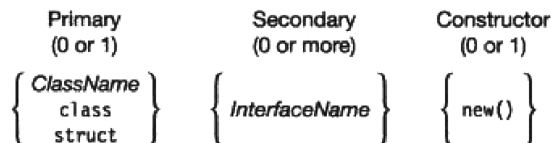



图19-9　如果类型参数有多个约束，它们必须遵照这个顺序

如下声明给出了一个where子句的示例：

```
class SortedList<S>
        where S: IComparable<S> { ... }

class LinkedList<M,N>
        where M : IComparable<M>
        where N : ICloneable     { ... }

class MyDictionary<KeyType, ValueType>
        where KeyType : IEnumerable,
        new()                    { ... }
```

## 19.8 泛型结构

与泛型类相似，泛型结构可以有类型参数和约束。泛型结构的规则和条件与泛型类是一样的。

例如，下面代码声明了一个叫做PieceOfData的泛型结构，它保存和获取一块数据，其中的类型在构建类型时被定义。Main创建了两个构造类型的对象——一个使用int，而另外一个使用string。

```
struct PieceOfData<T>                                  // 泛型结构
{
   public PieceOfData(T value) { _Data = value; }
   private T _Data;
   public  T Data
   {
      get { return _Data; }
      set { _Data = value; }
   }
}

class Program
{
   static void Main()                 构造类型
   {
      var intData    = new PieceOfData<int>(10);
```



```
        var stringData = new PieceOfData<string>("Hi there.");
                                      ↑
                                   构造类型

        Console.WriteLine("intData    = {0}", intData.Data);
        Console.WriteLine("stringData = {0}", stringData.Data);
    }
}
```

这段代码产生了如下的输出：

```
intData    = 10
stringData = Hi there.
```

## 19.9　泛型接口

泛型接口允许我们编写参数和接口成员返回类型是泛型类型参数的接口。泛型接口的声明和非泛型接口的声明差不多，但是需要在接口名称之后的尖括号中有类型参数。

例如，如下代码声明了叫做IMyIfc的泛型接口。

- 泛型类Simple实现了泛型接口。
- Main实例化了泛型类的两个对象，一个是int类型，另外一个是string类型。

```
              类型参数
                 ↓
interface IMyIfc<T>                        // 泛型接口
{
   T ReturnIt(T inValue);
}
           类型参数 泛型接口
               ↓       ↓
class Simple<S> : IMyIfc<S>                // 泛型类
{
   public S ReturnIt(S inValue)            // 实现接口
   { return inValue; }
}

class Program
{
   static void Main()
   {
      var trivInt    = new Simple<int>();
      var trivString = new Simple<string>();

      Console.WriteLine("{0}", trivInt.ReturnIt(5));
      Console.WriteLine("{0}", trivString.ReturnIt("Hi there."));
   }
}
```

这段代码产生了如下的输出：

```
5
Hi there.
```

### 19.9.1 使用泛型接口的示例

如下示例演示了泛型接口的两个额外的能力：

- 与其他泛型相似，实现不同类型参数的泛型接口是不同的接口。
- 我们可以在非泛型类型中实现泛型接口。

例如，下面的代码与前面的示例相似，但在这里，Simple是实现泛型接口的非泛型类。其实，它实现了两个IMyIfc的实例。一个实例使用int类型实例化，而另一个使string类型实例化。

```
interface IMyIfc<T>                              // 泛型接口
{
   T ReturnIt(T inValue);
}
                  源于同一泛型接口的两个不同接口
class Simple : IMyIfc<int>, IMyIfc<string>       // 非泛型类
{
   public int ReturnIt(int inValue)              // 实现int类型接口
   { return inValue; }

   public string ReturnIt(string inValue)        // 实现string类型接口
   { return inValue; }
}

class Program
{
   static void Main()
   {
      Simple trivInt    = new Simple();
      Simple trivString = new Simple();

      Console.WriteLine("{0}", trivInt.ReturnIt(5));
      Console.WriteLine("{0}", trivString.ReturnIt("Hi there."));
   }
}
```

这段代码产生了如下的输出：

```
5
Hi there.
```

### 19.9.2 泛型接口的实现必须唯一

实现泛型类型接口时，必须没有可能的类型实参组合会在类型中产生两个重复的接口。

例如，在下面的代码中，Simple类使用了两个IMyIfc接口的实例化。

- 第一个是构造类型，使用类型int进行实例化。
- 第二个有一个类型参数而不是实参。

这就引起了冲突，因为在第二个接口中如果把int用作类型实参，Simple就会有两个相同的接口——这是不允许的。

```
interface IMyIfc<T>
{
   T ReturnIt(T inValue);
}

                    两个接口
                  ↓          ↓
class Simple<S> : IMyIfc<int>, IMyIfc<S>     // 错误
{
   public int ReturnIt(int inValue)   // 实现第一个接口
   {
      return inValue;
   }

   public S ReturnIt(S inValue)       // 实现第二个接口
   {                                  // 如果它不是int类型的,
      return inValue;                 // 将和第一个接口一样
   }
}
```

**说明**　泛型接口的名字不会和非泛型冲突。例如，在前面的代码中我们还可以声明一个名称为ImyIfc的非泛型接口。

## 19.10　泛型委托

泛型委托和非泛型委托非常相似，除了决定哪种方法会被接受的类型参数之外。

- 要声明泛型委托，在委托名称后、委托参数列表之前的尖括号中放类型参数列表。

```
                      类型参数
                         ↓
delegate R MyDelegate<T, R>( T value );
         ↑                   ↑
       返回类型            委托形参
```

- 注意，在这里有两个参数列表：委托形参列表和类型参数列表。
- 类型参数的范围包括：
  - 返回值
  - 形参列表
  - 约束子句

如下代码给出了一个泛型委托的示例。在Main中，泛型委托MyDelegate使用string类型的实

参实例化，并且使用PrintString方法初始化。

```
delegate void MyDelegate<T>(T value);                  // 泛型委托

class Simple
{
   static public void PrintString(string s)        // 方法匹配委托
   {
      Console.WriteLine(s);
   }

   static public void PrintUpperString(string s)  // 方法匹配委托
   {
      Console.WriteLine("{0}", s.ToUpper());
   }
}

class Program
{
   static void Main( )
   {
      var myDel =                                       // 创建委托的实例
         new MyDelegate<string>(Simple.PrintString);
      myDel += Simple.PrintUpperString;                 // 添加方法

      myDel("Hi There.");                               // 调用委托
   }
}
```

这段代码产生了如下的输出：

```
Hi There.
HI THERE.
```

## 另一个泛型委托的示例

由于C# 3.0的LINQ特性在很多地方使用了泛型委托，我觉得在讲LINQ之前，有必要给出另外一个示例。在第21章中我会介绍LINQ以及更多有关它的泛型委托的内容。

如下代码声明了一个叫做Func的委托，它接受带有两个形参和一个返回值的方法。方法返回的类型被标识为TR，方法参数类型被标识为T1和T2。注意，委托返回的类型在泛型参数列表中的最后一个。

```
                            委托参数类型
                            ↓  ↓       ↓      ↓
public delegate TR Func<T1, T2, TR>(T1 p1, T2 p2);
                ↑                 ↑
class Simple          委托返回类型
```

```
{
    static public string PrintString(int p1, int p2)
    {
        int total = p1 + p2;
        return total.ToString();
    }
}

class Program
{
    static void Main()
    {
        var myDel =
            new Func<int, int, string>(Simple.PrintString);

        Console.WriteLine("Total: {0}", myDel(15, 13));
    }
}
```

这段代码产生了如下的输出：

```
Total: 28
```

## 19.11　泛型方法

与其他泛型不一样，方法是成员，不是类型。

泛型方法可以在泛型和非泛型类以及结构和接口中声明，如图19-10所示。

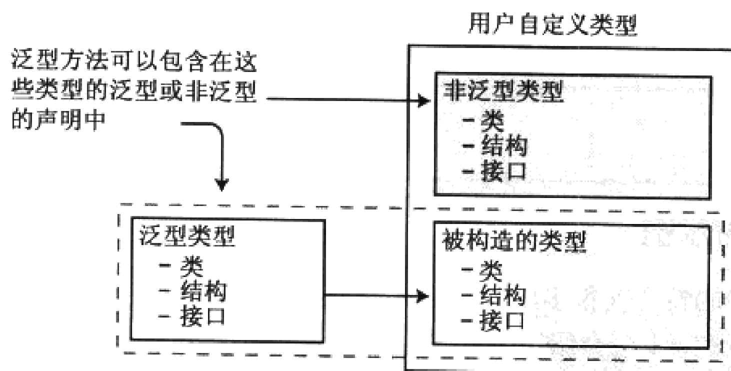



图19-10　泛型方法可以声明在泛型类型和非泛型类型中

### 19.11.1　声明泛型方法

泛型方法和其他泛型一样，有类型参数列表和可选的约束。

- 泛型方法和泛型委托相似，有两个参数列表：
  - 封闭在圆括号内的方法参数列表。
  - 封闭在尖括号内的类型参数列表。

- 要声明泛型方法，需要：
  - 在方法名称和方法参数列表之前放类型参数列表。
  - 在方法参数列表后放可选的约束子句。

```
                类型参数列表          约束子句
                    ↓                   ↓
public void PrintData<S, T> ( S p ) where S: Person
{                               ↑
   ...                      方法参数列表
}
```

**说明** *记住，类型参数列表在方法名称后，在方法参数列表之前。*

## 19.11.2 调用泛型方法

要调用泛型方法，应该在方法调用时提供类型实参，如下所示：

```
           类型参数
              ↓
MyMethod<short, int>();
MyMethod<int, long >();
```

图19-11演示了一个叫做DoStuff的泛型方法的声明，它接受两个类型参数。下面是使用不同参数类型的两次方法调用。每一次调用产生了不同版本的方法，如图19-11右边所示。

```
void DoStuff< T1, T2 >( )              void DoStuff <short,int >( ) {
{                                         short SomeVar;
   T1 SomeVar;                            int   OtherVar;
   T2 OtherVar;                             ...
     ...                               }
}
                                       void DoStuff <int,long >( ) {
   ...                                    int   SomeVar;
DoStuff< short, int > ( );                long  OtherVar;
                                            ...
DoStuff< int, long > ( );              }
```

图19-11 有两个实例的泛型方法

### 推断类型

如果我们为方法传入参数，编译器有时可以从方法参数中推断出泛型方法的类型形参中用到的那些类型。这样就可以使方法调用更简单，可读性更强。

例如，下面的代码声明了MyMethod，它接受了一个与类型参数同类型的方法参数。

```
public void MyMethod <T> (T myVal) { ... }
                      ↑   ↑
                   两个都是T类型
```

如下代码所示，如果我们使用int类型的变量调用MyMethod，方法调用中的类型参数的信息就多余了，因为编译器可以从方法参数中得知它是int。

```
int MyInt = 5;
MyMethod <int> (MyInt);
            ↑      ↑
            两个都是int
```

由于编译器可以从方法参数中推断类型参数，我们可以省略类型参数和调用中的尖括号，如下所示。

```
MyMethod(MyInt);
```

### 19.11.3　泛型方法的示例

如下的代码在一个叫做Simple的非泛型类中声明了一个叫做ReverseAndPrint的泛型方法。这个方法把任意类型的数组作为其参数。Main声明了三个不同的数组类型，然后使用每一个数组调用方法两次。第一次使用特定数组调用了方法，并显式使用类型参数，而第二次让编译器推断类型。

```
class Simple                                              //非泛型类
{
   static public void ReverseAndPrint<T>(T[] arr)         //泛型方法
   {
      Array.Reverse(arr);
      foreach (T item in arr)                             //使用类型参
         Console.Write("{0}, ", item.ToString());
      Console.WriteLine("");
   }
}

class Program
{
   static void Main()
   {
      // Create arrays of various types.
      var intArray    = new int[]    { 3, 5, 7, 9, 11 };
      var stringArray = new string[] { "first", "second", "third" };
      var doubleArray = new double[] { 3.567, 7.891, 2.345 };

      Simple.ReverseAndPrint<int>(intArray);              // 调用方法
      Simple.ReverseAndPrint(intArray);                   // 引用类型并调用

      Simple.ReverseAndPrint<string>(stringArray);        // 调用方法
      Simple.ReverseAndPrint(stringArray);                // 引用类型并调用

      Simple.ReverseAndPrint<double>(doubleArray);        // 调用方法
      Simple.ReverseAndPrint(doubleArray);                // 引用类型并调用
```

```
    }
}
```

这段代码产生了如下的输出：

```
11 9 7 5 3
3 5 7 9 11
third second first
first second third
2.345 7.891 3.567
3.567 7.891 2.345
```

## 19.12 扩展方法和泛型类

在第7章中，我们介绍了扩展方法，它也可以和泛型类结合使用。它允许我们将类中的静态方法关联到不同的泛型类上，还允许我们像调用类构造实例的实例方法一样来调用方法。

和非泛型类一样，泛型类的扩展方法：

- 必须声明为static；
- 必须是静态类的成员；
- 第一个参数类型中必须有关键词this，后面是扩展的泛型类的名字。

如下代码给出了一个叫做Print的扩展方法，扩展了叫做Holder<T>的泛型类。

```
static class ExtendHolder
{
    public static void Print<T>(this Holder<T> h)
    {
        T[] vals = h.GetValues();
        Console.WriteLine("{0},\t{1},\t{2}", vals[0], vals[1], vals[2]);
    }
}

class Holder<T>
{
    T[] Vals = new T[3];

    public Holder(T v0, T v1, T v2)
    { Vals[0] = v0; Vals[1] = v1; Vals[2] = v2; }

    public T[] GetValues() { return Vals; }
}

class Program
{
    static void Main(string[] args) {
        var intHolder    = new Holder<int>(3, 5, 7);
        var stringHolder = new Holder<string>("a1", "b2", "c3");
        intHolder.Print();
```

```
        stringHolder.Print();
    }
}
```

这段代码产生了如下的输出：

```
3,       5,       7
a1,      b2,      c3
```

# 第 20 章 枚举数和迭代器

20



本章内容



## 20.1 枚举数和可枚举类型

在第14章中，我们已经知道可以使用foreach语句来遍历数组中的元素。在本章中，我们会进一步探讨数组，来看看为什么它们可以被foreach语句处理。我们还会研究如何为用户自定义的类增加这个功能。在本章的后面部分，我们会讨论迭代器的使用。

### 20.1.1 使用 **foreach** 语句

当我们为数组使用foreach语句时，这个语句为我们依次取出了数组中的每一个元素，允许我们读取它的值。

例如，如下的代码声明了一个有4个元素的数组，然后使用foreach来循环打印这些项的值：

```
int[] arr1 = { 10, 11, 12, 13 };                // Define the array.

foreach (int item in arr1)                      // Enumerate the elements.
   Console.WriteLine("Item value:  {0}", item);
```

这段代码产生了如下的输出：

```
Item value:  10
Item value:  11
Item value:  12
Item value:  13
```

为什么数组可以这么做？看上去很神奇。原因是数组可以按需提供一个叫做枚举数（enumerator）的对象。枚举数可以依次返回请求的数组的元素。枚举数“知道”项的次序并且跟踪它在序列中的位置，然后返回请求的当前项。

对于有枚举数的类型而言，必须有一个方法来获取它们。在.NET中获取一个对象枚举数的标准方法是调用对象的`GetEnumerator`方法。实现`GetEnumerator`方法的类型叫做可枚举类型（enumerable type或enumerable）。数组是可枚举类型。

图20-1演示了可枚举类型和枚举数之间的关系。

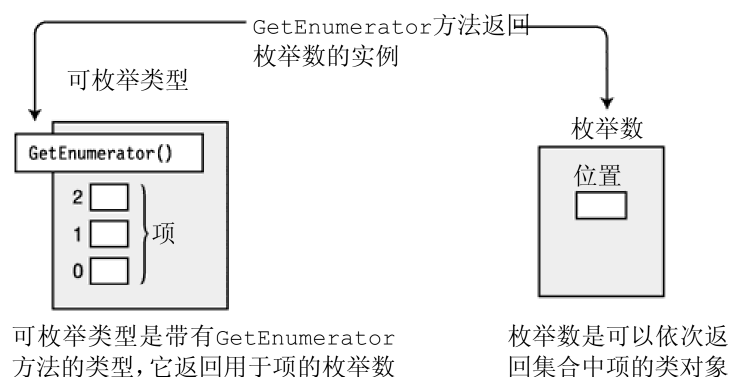



图20-1　枚举数和可枚举类型概览

`foreach`结构被设计用来和可枚举类型一起使用。只要给它的遍历对象是可枚举类型，比如数组，它就会执行如下行为：

- 通过调用`GetEnumerator`方法获取对象的枚举数。
- 从枚举数中请求每一项并且把它作为迭代变量（iteration variable），代码可以读但不可以改变。

必须是可枚举类型

```
foreach( Type VarName in EnumerableObject )
{
   ...
}
```

## 20.1.2　枚举数类型

枚举数一共有三种。它们的工作原理本质上是一样的，只是有一些细小的区别。我会讨论每一种类型。我们可以使用以下方式实现枚举数：

- `IEnumerator/IEnumerable`接口——叫做非泛型接口形式。

- IEnumerator<T>/IEnumerable<T>接口——叫做泛型接口形式。
- 不使用接口形式。

## 20.2 使用 IEnumerator 接口

这部分内容会先来介绍前面列表中的非泛型接口形式。这种形式的枚举数是实现IEnumerator接口的类。之所以叫做非泛型是因为它没有使用C#泛型。

IEnumerator接口包含三个函数成员：Current、MoveNext以及Reset。

- Current返回序列中当前位置项的属性。
  - 它是只读属性。
  - 它返回object类型的引用，所以可以返回任何类型。
- MoveNext是把枚举数位置前进到集合中下一项的方法。它也返回布尔值，指示新的位置是有效位置或已经超过了序列的尾部。
  - 如果新的位置是有效的，方法返回true。
  - 如果新的位置是无效的（比如到达了尾部），方法返回false。
  - 枚举数的原始位置在序列中的第一项之前。MoveNext必须在第一次使用Current之前使用，否则CLR会抛出一个InvalidOperationException异常。
- Reset方法把位置重置为原始状态。

图20-2左部显示了三个项的集合，在右部显示了枚举数。在图20-2中，枚举数是一个叫做ArrEnumerator类的实例。

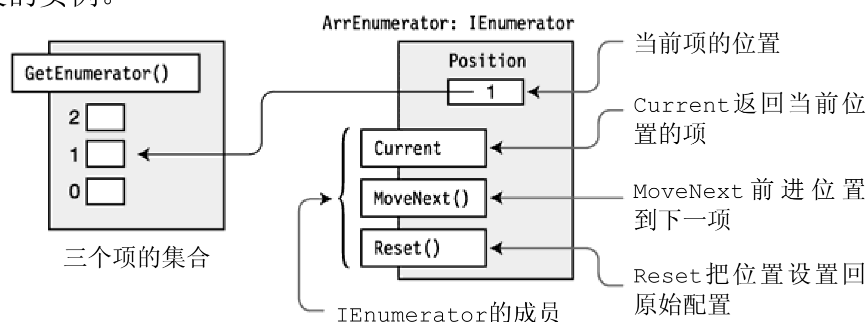



图20-2 小集合的枚举数

枚举数类通常被声明为类中的嵌套类。嵌套类是声明在另外一个类声明中的类，第25章描述了嵌套类的一些细节。

枚举数与序列中的当前项保持联系的方式完全取决于实现。可以通过对象引用、索引值或其他方式来实现。对于数组来说，就使用项的索引。

图20-3演示了有三个项的集合的枚举数的状态。这些状态标记了1到5。

- 注意，在状态1中，枚举数的原始位置是-1（也就是在集合的第一个元素之前）。
- 状态的每次切换都由MoveNext进行，它提升了序列中的位置。每次调用MoveNext时，状态1到状态4都返回true，然而，在从状态4到状态5的过程中，位置最终超过了集合的最后一项，所以方法返回false。

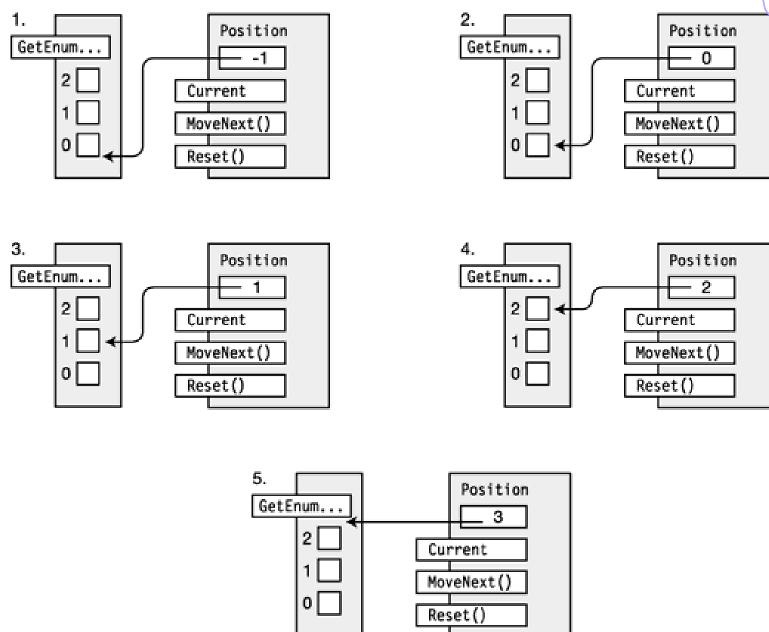



图20-3　枚举数状态

- 在最后一个状态中，任何进一步的调用MoveNext总是会返回false。

有了集合的枚举数，我们就可以使用MoveNext和Current成员来模仿foreach循环遍历集合中的项。我们已经知道了，例如数组就是可枚举类型，所以下面的代码手动做foreach语句自动做的事情。输出和使用foreach循环的输出一样。

```
static void Main()
{
   int[] MyArray = { 10, 11, 12, 13 };

   IEnumerator ie = MyArray.GetEnumerator();      // 获取枚举数
   while ( ie.MoveNext() )                         // 移到下一项
   {
      int i = (int) ie.Current;                    // 获取当前项
      Console.WriteLine("{0}", i);                 // 输出
   }
}
```

这段代码产生了如下的输出：

```
10
11
12
13
```

## 声明 **IEnumerator** 的枚举数

要创建非泛型接口的枚举数类，我们必须声明实现IEnumerator接口的类。IEnumerator接口有如下的特性：

- 它是System.Collection命名空间的成员。
- 它包含三个方法Current、MoveNext和Reset。

如下代码演示了非泛型枚举数类的框架。这里没有显示是如何保持位置的。注意，Current返回object的引用。

```
using System.Collections;                         // 包含命名空间

class MyEnumerator: IEnumerator
{       返回object的引
              ↓
   public object Current  { get; }                // Current

   public bool MoveNext() { ... }                 // MoveNext

   public void Reset()    { ... }                 // Reset
      ...
}
```

例如，下面代码实现了一个列出颜色名数组的枚举数类：

```
using System.Collections;

class ColorEnumerator: IEnumerator
{                               ↑
   string[] Colors;     实现IEnumerator
   int Position = -1;

   public object Current                                   // Current
   {
      get { return Colors[Position]; }
   }

   public bool MoveNext()                                  // MoveNext
   {
      if (Position < Colors.Length - 1)
      {
         Position++;
         return true;
      }
      else
         return false;
   }

   public void Reset()                                     // Reset
   {
      Position = -1;
   }

   public ColorEnumerator(string[] theColors)              // 构造函数
```

```
    {
       Colors = new string[theColors.Length];
       for (int i = 0; i < theColors.Length; i++)
          Colors[i] = theColors[i];
    }
}
```

## 20.3　IEnumerable 接口

IEnumerable接口只有一个成员——GetEnumerator方法，它返回对象的枚举数。

图20-4演示了一个有三个要枚举项的类MyClass，通过实现GetEnumerator方法来实现IEnumerable接口。

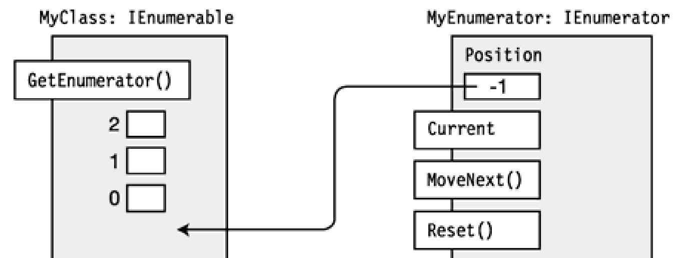



图20-4　GetEnumerator方法返回类的一个枚举数对

如下代码演示了可枚举类的声明形式：

```
using System.Collections;
                    实现IEnumerable接
                          ↓
class MyClass : IEnumerable
{
   public IEnumerator GetEnumerator { ... }
      ...     ↑
}   返回IEnumerator类型的对
```

如下的代码给出了一个使用之前示例中的ColorEnumerator枚举数类的示例。要知道，ColorEnumerator实现了IEnumerator。

```
using System.Collections;

class MyColors: IEnumerable
{
   string[] Colors = { "Red", "Yellow", "Blue" };

   public IEnumerator GetEnumerator()
   {
      return new ColorEnumerator(Colors);
   }                  ↑
}                枚举数类的实例
```

### 使用 IEnumerable 和 IEnumerator 的示例

把MyColors和ColorEnumerator示例放在一起，我们可以添加一个叫做Program的类，其中有一个Main方法用来创建MyColors的实例，并在foreach中使用。

```
using System;
using System.Collections;
```

```
namespace ColorCollectionEnumerator
{
   class ColorEnumerator: IEnumerator
   {
      string[] Colors;
      int Position = -1;

      public ColorEnumerator(string[] theColors)          // 构造函数
      {
         Colors = new string[theColors.Length];
         for (int i = 0; i < theColors.Length; i++)
            Colors[i] = theColors[i];
      }

      public object Current                               // Current
      {
         get { return Colors[Position]; }
      }

      public bool MoveNext()                              // MoveNext
      {
         if (Position < Colors.Length - 1)
         {
            Position++;
            return true;
         }
         else
            return false;
      }

      public void Reset()                                 // Reset
      { Position = -1; }
   }
   class MyColors: IEnumerable
   {
      string[] Colors = { "Red", "Yellow", "Blue" };

      public IEnumerator GetEnumerator()
      {
         return new ColorEnumerator(Colors);
      }
   }

   class Program
   {
      static void Main()
      {
         MyColors mc = new MyColors();
         foreach (string color in mc)
            Console.WriteLine("{0}", color);
```

```
            }
        }
    }
```

这段代码产生了如下的输出：

```
Red
Yellow
Blue
```

## 20.4　不实现接口的枚举数

我们已经知道如何使用IEnumerable和IEnumerator接口来创建可枚举类型和枚举数，但是这种方法有几个缺点。

首先，由Current返回的对象是object类型的。对于值类型而言，在由Current返回之前必须装箱成object。在从Current获取之后，又必须再一次拆箱。如果需要操作大量的数据，会带来严重的性能问题。

非泛型接口方法的另外一个缺点是失去了类型安全。值被作为对象来枚举，所以可以是任何类型。这就消除了编译时的类型检测。

我们可以通过对枚举数和可枚举类型类的声明做如下改变来解决这个问题。

- 对于枚举数类：
  - 不要继承自IEnumerator。
  - 像以前一样实现MoveNext。
  - 像以前一样实现Current，把返回类型设置为和枚举的项一样。
  - 不需要实现Reset。
- 对于可枚举类：
  - 不要继承自IEnumerable。
  - 像以前一样实现GetEnumerator，设置返回值为枚举数类。

图20-5展示了它们的区别。左边是非泛型接口代码，右边是非接口代码。有了这样的修改，foreach语句可以完美处理集合而且还没有前面列出的一些缺点。

```
class SibEnumerator : IEnumerator
{
   ...
   public object Current
      { get { ... } }

   public bool MoveNext()
   { ... }

   public void Reset()
   { ... }
}

class Siblings : IEnumerable
{
   ...
   public IEnumerator GetEnumerator()
   { ... }
}
```

```
class SibEnumerator
{
   ...
   public string Current
      { get { ... } }

   public bool MoveNext()
   { ... }

}

class Siblings
{
   ...
   public SibEnumerator GetEnumerator()
   { ... }
}
```

图20-5　比较基于接口的和非基于接口的枚举数

对于非接口的枚举数的实现来说，一个可能的问题就是其他程序集的类型可能希望使用接口方法来实现枚举。如果这些类型尝试使用接口转换来获取类对象的枚举，可能就会找不到。

要解决这个问题，我们可以在同一个类中实现两种形式。也就是说，在类级别创建Current、MoveNext、Reset和GetEnumerator的实现，并且也为它们创建显式接口实现。有了这两组实现，foreach和其他结构会调用类型安全的、有效的实现，而其他结构可以调用显式接口实现。

## 20.5　泛型枚举接口

第三种形式的枚举数是使用泛型接口IEnumerable<T>和IEnumerator<T>。它们被叫做泛型是因为使用了C#泛型，使用方式和非泛型形式差不多。当然，这两者之间的差别如下所示。

- 对于非泛型接口形式：
  - IEnumerable接口的GetEnumerator方法返回实现IEnumerator枚举数类的实例。
  - 实现IEnumerator的类实现了Current属性，它返回object的引用，然后我们必须把它转化为实际类型的对象。
- 对于泛型接口形式：
  - IEnumerable<T>接口的GetEnumerator方法返回实现IEnumator<T>的枚举数类的实例。
  - 实现IEnumerator<T>的类实现了Current属性，它返回实际类型的对象，而不是object基类的引用。

需要重点注意的是，非泛型接口的实现不是类型安全的。它们返回object类型的引用，然后需要转化为实际类型，而泛型接口的枚举数是类型安全的，它返回实际类型的引用。

## 20.6　IEnumerator<T>接口

IEnumerator<T>接口使用泛型来返回实际的类型，而不是object类型的对象。

IEnumerator<T>接口从另外两个接口继承——非泛型IEnumerable接口和IDisposable接口。所以，它肯定实现了它们的成员。

- 我们已经知道了非泛型接口IEnumerable和它的三个成员。
- IDisposable接口只有一个叫做Dispose的类型为void的无参方法，它可以用于释放由类占据的非托管资源（在第6章中介绍过Dispose方法）。
- IEnumerator<T>接口本身只有一个Current方法，它返回衍生类型的项——不是object类型的项。
- 由于IEnumerator<T>和IEnumerator都有一个叫做Current的成员，我们应该显式实现IEnumerator版本，然后在类中实现泛型版本，如图20-6所示。

图20-6演示了接口的实现。

实现接口的类的声明和下面的代码差不多，T是有枚举数返回的类型。

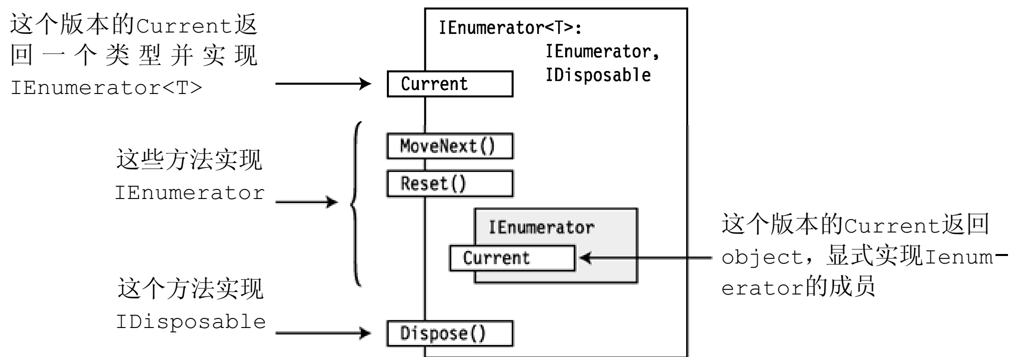



图20-6 实现IEnumerator<T>接口

```
using System.Collections;
using System.Collections.Generic;

class MyGenEnumerator: IEnumerator< T >
{
   public T Current  { get; }                    // IEnumerator<T>——Current
                显式实现
                  ↓
   object IEnumerator.Current { get { ... } }    // IEnumerator——Current

   public bool MoveNext() { ... }                // IEnumerator——MoveNext

   public void Reset()    { ... }                // IEnumerator——Reset

   public void Dispose()  { ... }                // IDisposable——Dispose
      ...
}
```

例如，如下代码使用泛型枚举数接口来实现ColorEnumerator实例：

```
using System.Collections;
using System.Collections.Generic;  Substitute type for T

class ColorEnumerator : IEnumerator<string>
{
   string[] Colors; int Position = -1;

        返回衍生类型
   public string Current                         // Current——泛型
   {
      get { return Colors[Position]; }
   }

              显式实现
   object IEnumerator.Current                    // Current——非泛型
   {
      get { return Colors[Position]; }
   }

   public bool MoveNext()                        // MoveNext
   {
      if (Position < Colors.Length - 1)
      {
         Position++;
         return true;
      }
      else
         return false;
   }

   public void Reset()                           // Reset
   { Position = -1; }

   public void Dispose() { }

   public ColorEnumerator(string[] colors)       // 构造函数
   {
      Colors = new string[colors.Length];
      for (int i = 0; i < colors.Length; i++)
         Colors[i] = colors[i];
   }
}
```

## 20.7 IEnumerable<T>接口

泛型IEnumerable<T>接口与IEnumerable的非泛型版本很相似。泛型版本从IEnumerable继承，所以也必须实现IEnumerable接口。

- 与IEnumerable差不多，泛型版本也包含了一个方法——GetEnumerator。然而，这个版本的GetEnumeratorfanui实现泛型IEnumerator<T>接口的类对象。
- 由于类必须实现两个GetEnumerator方法，我们需要显式实现非泛型版本，并在类中实现泛型版本，如图20-7所示。

图20-7演示了接口的实现

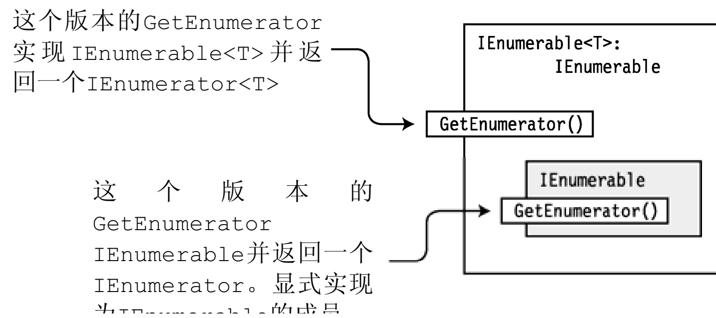



图20-7 实现IEnumerable<T>接口

如下代码给出了实现泛型接口的模式。T是由枚举数返回的类型。

```
using System.Collections;
using System.Collections.Generic;

class MyGenEnumerable: IEnumerable<T>
{
   public IEnumerator<T> GetEnumerator() { ... }     // IEnumerable<T> 版本
                    显式实现
                       ↓
   IEnumerator IEnumerable.GetEnumerator() { ... }   // IEnumerable 版本
      ...
}
```

例如，如下代码演示了泛型可枚举接口的使用：

```
using System.Collections;
using System.Collections.Generic;
                        为T替换实际类型
                              ↓
class MyColors : IEnumerable<string>
{
   string[] Colors = { "Red", "Yellow", "Blue" };
                  为T替换实际类型
                        ↓
   public IEnumerator<string> GetEnumerator()        // IEnumerable<T> 版本
   {
      return new ColorEnumerator(Colors);
   }
                    显式实现
                       ↓
   IEnumerator IEnumerable.GetEnumerator()           // IEnumerable 版本
   {
      return new ColorEnumerator(Colors);
   }
}
```

## 20.8 迭代器

可枚举类型类和枚举数在.NET集合类中被广泛使用，所以知道它们如何工作很重要。但是，既然我们已经知道如何创建自己的可枚举类以及枚举数了，我们可能会很高兴听到，C#从2.0版本开始提供了更简单的创建枚举数和可枚举类型的方式。其实，编译器会为我们创建它们。这种结构叫做迭代器。我们可以把手动编码的可枚举类型和枚举数替换为由迭代器生成的可枚举类型和枚举数。

在解释细节之前，让我们先来看两个示例。下面的方法声明实现了一个产生和返回枚举数的迭代器。

- 迭代器返回一个泛型枚举数，该枚举数返回三个`string`类型的项。
- `yield return`语句声明这是枚举中的下一项。

```
          返回泛型枚举数
                ↓
public IEnumerator<string> BlackAndWhite()           // 版本 1
{
   yield return "black";                             // yield return
   yield return "gray";                              // yield return
   yield return "white";                             // yield return
}
```

下面的方法声明了另一个版本，并输出了相同的结果：

```
          返回泛型枚举数
public IEnumerator<string> BlackAndWhite()            // 版本 2
{
   string[] TheColors = { "black", "gray", "white" };

   for (int i = 0; i < TheColors.Length; i++)
      yield return TheColors[i];                      // yield return
}
```

到现在为止，我还没有解释过yield return语句。但是如果仔细看代码，你可能会觉得代码有一些奇怪。但它又很正确，那么yield return语句究竟做什么呢？

例如，在第一个版本中，如果方法在第一个yield return语句处返回，那么后两条语句永远不会到达。如果没有在第一条语句中返回，而是继续后面的代码，在这些值上发生了什么呢？在第二个版本中，如果循环主体中的yield return语句在第一个迭代中返回，循环永远不会获得其他的后续迭代。

除此之外，枚举数不会一次返回所有元素——每次访问Current属性时便返回一个新值。那么是怎么为我们实现枚举数的呢？很明显，该代码与之前给出的代码很不相同。

## 20.8.1 迭代器块

迭代器块是有一个或多个yield语句的代码块。下面三种类型的代码块中的任意一种都可以是迭代器块：

- 方法主体
- 访问器主体
- 运算符主体

迭代器块被认为与其他代码块不同。其他块包含的语句被当作是命令式的。也就是说，代码块的第一个语句被执行然后是后面的语句，最后控制离开块。

另一方面，迭代器块不是需要在同一时间执行的一序列命令式命令，而是描述了希望编译器为我们创建的枚举数类的行为。迭代器块中的代码描述了如何枚举元素。

迭代器块有两个特殊语句：

- yield return语句执行了序列中返回的下一项。
- yield break语句指定在序列中没有更多项。

编译器接受到有关如何枚举项的描述后使用它来构建包含所有需要的方法和属性实现的枚举数类。结果类被嵌套包含在迭代器声明的类中。图20-8 在左边给出了代码，在右边演示了结果对象。注意，编译器自动为我们做了很多事情。

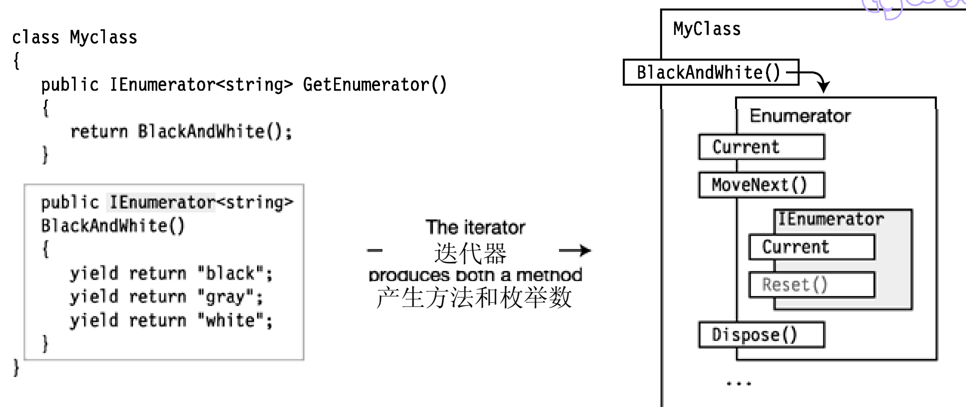



图20-8 产生枚举数的迭代器

## 20.8.2 使用迭代器来创建枚举数

下面代码演示了如何使用迭代器来创建可枚举类型的类。

- ❑ 图20-8演示的`MyClass`使用迭代器方法`BlackAndWhite`来产生迭代数。
- ❑ `MyClass`还实现了`GetEnumerator`方法，它调用`BlackAndWhite`并且返回`BlackAndWhite`给它的枚举数。
- ❑ 注意`Main`方法，由于类是可枚举类型，我们在`foreach`语句中直接使用了类的实例。

```
class MyClass
{
   public IEnumerator<string> GetEnumerator()
   {
      return BlackAndWhite();                    // 返回枚举数
   }
              返回枚举数
                 ↓
   public IEnumerator<string> BlackAndWhite()   // 迭代器
   {
      yield return "black";
      yield return "gray";
      yield return "white";
   }
}
```

```
class Program
{
   static void Main()
   {
      MyClass mc = new MyClass();

                                ↓
      foreach (string shade in mc)
         Console.WriteLine(shade);
   }
}
```

这段代码产生了如下的输出：

```
black
gray
white
```

### 20.8.3 使用迭代器来创建可枚举类型

之前的示例创建的类包含两部分：产生枚举数的迭代器以及返回枚举数的GetEnumerator方法。在这个示例中，迭代器被用于创建可枚举类型，而不是枚举数。与之前的示例相比，本例有一些重要的不同：

- 在之前的示例中，BlackAndWhite迭代器方法返回IEnumerator<string>，MyClass类通过返回由BlackAndWhite返回的对象来实现GetEnumerator方法。
- 在本例中，BlackAndWhite迭代器方法返回IEnumerable<string>而不是Ienumerator <string>。因此，MyClass首先调用BlackAndWhite方法获取它的可枚举类型对象，然后调用对象的GetEnumerator方法来获取它的结果，从而实现GetEnumerator方法。
- 注意，在Main的foreach语句中，我们可以使用类的实例，也可以直接调用BlackAndWhite方法，因为它返回的是可枚举类型。两种方法如下：

```
class MyClass {
   public IEnumerator<string> GetEnumerator()
   {
      IEnumerable<string> myEnumerable = BlackAndWhite(); // 获取枚举数
      return myEnumerable.GetEnumerator();                // 获取枚举数
   }
              使用MyClass的实例
                    ↓
   public IEnumerable<string> BlackAndWhite()
   {
      yield return "black";
      yield return "gray";
      yield return "white";
   }
}

class Program {
   static void Main()
   {
      MyClass mc = new MyClass();
                             使用类对象
                                 ↓
      foreach (string shade in mc)
         Console.Write("{0}  ", shade); 使用类枚举数方法
                                              ↓
      foreach (string shade in mc.BlackAndWhite())
         Console.Write("{0}  ", shade);
   }
}
```

这段代码产生了如下的输出：

```
black  gray  white  black  gray  white
```

## 20.9　常见迭代器模式

前面两节的内容显示了，我们可以创建迭代器来返回可枚举类型或枚举数。图20-9总结了如何使用普通迭代器模式。

- 当我们实现返回枚举数的迭代器时，必须通过实现`GetEnumerator`来让类可以被枚举，它返回由迭代器返回的枚举数。如图20-9中左部分所示。

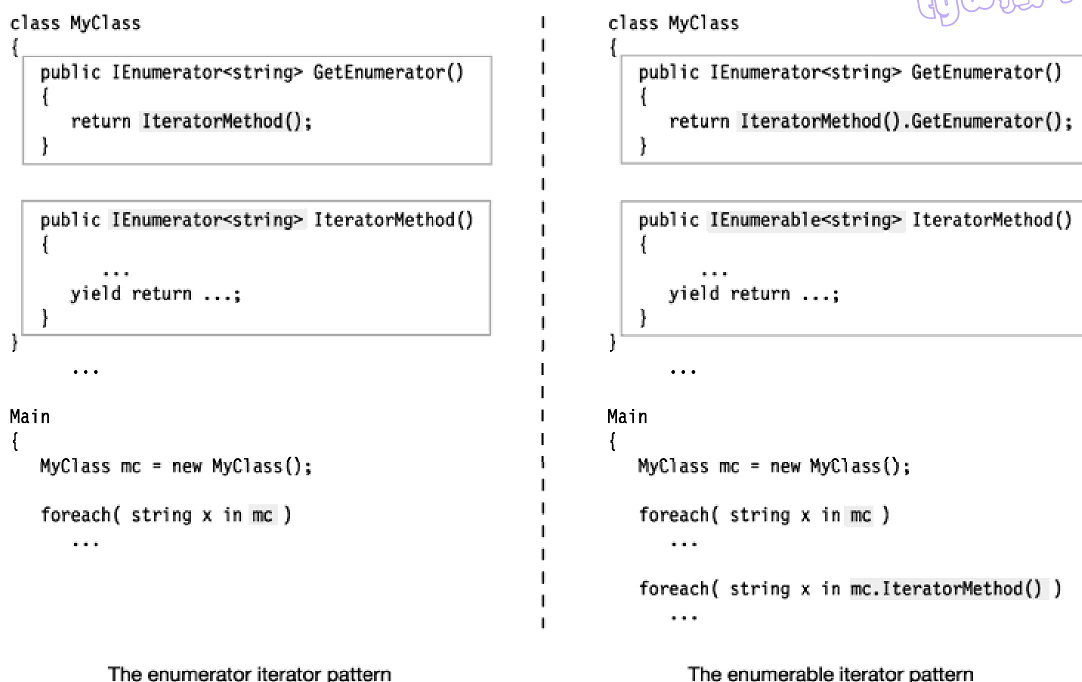



图20-9　常见迭代器模式

- 如果我们在类中实现的迭代器返回可枚举类型，我们可以让类实现或不实现GetEnumerator来让类本身可被枚举或不可被枚举。
  - 如果实现GetEnumerator，让它调用迭代器方法以获取自动生成的实现IEnumerable的类实例。然后，从IEnumerable对象返回由GetEnumerator创建的枚举数，如图20-9右边所示。
  - 如果通过不实现GetEnumerator使类本身不可被枚举，仍然可以使用由迭代器返回的可枚举类，只需要直接调用迭代器方法，如图20-9中右边第二个foreach语句所示。

## 20.10　产生可枚举类型和枚举数

之前的示例使用的迭代器返回IEnumerator<T>或IEnumerable<T>。我们还可以创建迭代器来返回非泛型的版本。可以指定如下的返回类型：

- IEnumerator<T>（泛型——替代T为实际类型）；
- IEnumerable<T>（泛型——替代T为实际类型）；
- IEnumerator（非泛型）；
- IEnumerable（非泛型）。

对于这两个枚举数类型，编译器生成包含非泛型或泛型枚举数的嵌套类，它的行为由迭代器块指定。

对于两个可枚举类型，它做得更多。它产生一个既是可枚举类型又是枚举数的嵌套类。因此，这个类实现了枚举数和GetEnumerator方法。注意，GetEnumerator被作为嵌套类的一部分来

实现，而不是封闭类的一部分。

图20-10演示了由上例中的可枚举类型迭代器产生的泛型可枚举类型。

- 迭代器的代码在图的左边，并且可以看到返回类型是IEnumerable<string>。
- 图的右边显示嵌套类型实现了IEnumerater<string>和IEnumerable<string>。

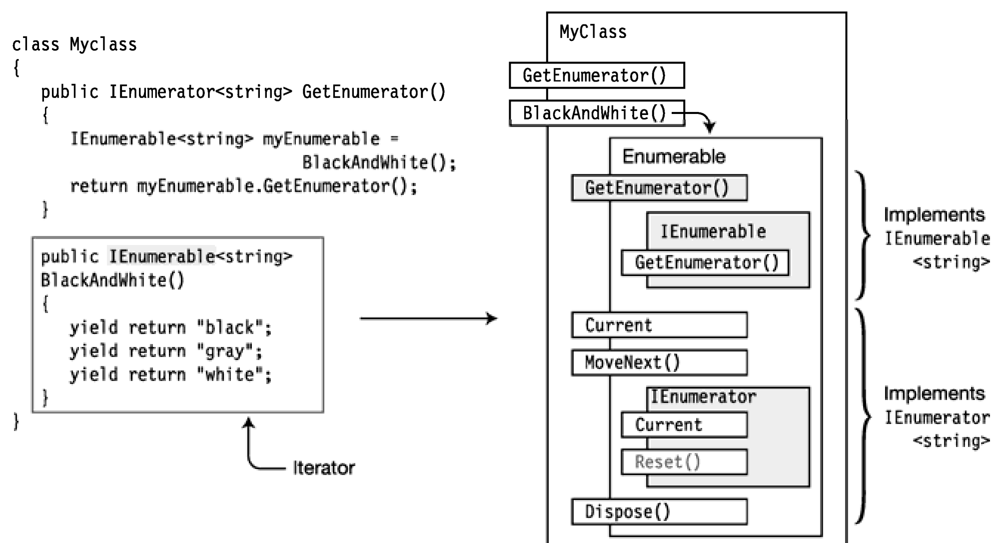



图20-10　编译器产生了一个既是可枚举类型又是枚举数的类。它也产生了一个返回类对象的方法

## 20.11　产生多个可枚举类型

在下面的示例中，ColorCollection类有两个可枚举类型的迭代器——一个以正序进行枚举，而另一个以逆序进行枚举。注意，尽管它有两个方法返回可枚举类型，类本身不是可枚举类型，因为它没有实现GetEnumerator。

```
using System;
using System.Collections.Generic;                    // 需要此命名空间

namespace ColorCollectionIterator
{
   class ColorCollection
   {
      string[] Colors={"Red", "Orange", "Yellow", "Green", "Blue", "Purple"};

      public IEnumerable<string> Forward() {        // 可枚举类型的迭代器
         for (int i = 0; i < Colors.Length; i++)
            yield return Colors[i];
      }

      public IEnumerable<string> Reverse() {        // 可枚举类型的迭代器
         for (int i = Colors.Length - 1; i >= 0; i--)
            yield return Colors[i];
      }
   }

   class Program
   {
      static void Main()
      {
         ColorCollection cc = new ColorCollection();
                              为foreach语句返回可枚举类型
                                        ↓
         foreach (string color in cc.Forward())
            Console.Write("{0} ", color);
         Console.WriteLine("");
                              为foreach语句返回可枚举类型
                                        ↓
         foreach (string color in cc.Reverse())
            Console.Write("{0} ", color);
         Console.WriteLine("");

         // Skip the foreach and manually use the enumerable and enumerator.
         IEnumerable<string> ieable = cc.Reverse();
         IEnumerator<string> ieator = ieable.GetEnumerator();
         while (ieator.MoveNext())
            Console.Write("{0} ", ieator.Current);
         Console.WriteLine("");
      }
   }
}
```

这段代码产生了如下的输出：

```
Red Orange Yellow Green Blue Purple
Purple Blue Green Yellow Orange Red
Purple Blue Green Yellow Orange Red
```

## 20.12　产生多个枚举数

之前的示例使用迭代器来产生具有两个可枚举类型的类。本例演示两个方面的内容：第一，它使用迭代器来产生具有两个枚举数的类；第二，它演示了迭代器如何能实现为属性而不是方法。

这段代码声明了两个属性来定义两个不同的枚举数。GetEnumerator方法根据ColorFlag布尔变量的值返回两个枚举数的一个或另外一个。如果ColorFlag为true，则返回Colors枚举数；否则，返回BlackAndWhite枚举数。

```
class MyClass: IEnumerable<string>
{
   bool ColorFlag = true;

   public MyClass(bool flag)
   {
      ColorFlag = flag;
   }

   IEnumerator<string> BlackAndWhite        // 属性——枚举数迭代器
   {
      get {
         yield return "black";
         yield return "gray";
         yield return "white";
      }
   }

   IEnumerator<string> Colors               // 属性——枚举数迭代器
   {
      get {
         string[] TheColors = { "blue", "red", "yellow" };
         for (int i = 0; i < TheColors.Length; i++)
            yield return TheColors[i];
      }
   }

   public IEnumerator<string> GetEnumerator()  // GetEnumerator
   {
      return ColorFlag
```

```
                    ? Colors                    // 返回Colors的枚举数
                    : BlackAndWhite;            // 返回BlackAndWhite的枚举数
    }

    System.Collections.IEnumerator
    System.Collections.IEnumerable.GetEnumerator()
    {
       return ColorFlag
                    ? Colors                    // 返回Colors的枚举数
                    : BlackAndWhite;            // 返回BlackAndWhite的枚举数
    }
}

class Program
{
    static void Main()
    {
       MyClass mc1 = new MyClass( true );          // 调用参数为true的构造函数
       foreach (string s in mc1)
          Console.Write("{0}  ", s);
       Console.WriteLine("");

       MyClass mc2 = new MyClass( false );        // 调用参数为false的构造函数
       foreach (string s in mc2)
          Console.Write("{0}  ", s);
       Console.WriteLine("");
    }
}
```

这段代码产生了如下的输出：

```
blue   red   yellow
black   gray   white
```

## 20.13　迭代器实质

如下是需要了解的有关迭代器的其他重要事项：

- 迭代器需要System.Collections.Generic命名空间，因此我们需要使用using指令包含它。
- 在编译器生成的枚举数中，Reset方法没有实现。而它是接口需要的方法，因此调用实现时总是抛出System.NotSupportedException异常。注意，在图20-8中Reset方法被显示为灰色。

在后台，由编译器生成的枚举数类是有4个状态的状态机。

- Before：首次调用MoveNext的初始状态。
- Running：调用MoveNext后进入这个状态。在这个状态中，枚举数检测并设置下一项的位置。在遇到yield return、yield break或在迭代器体结束时，退出状态。

- Suspended：状态机等待下次调用MoveNext的状态。
- After：没有更多项可以枚举。

如果状态机在before或suspended状态，并且有一次MoveNext方法调用，它就转到了running状态。在running状态中，它检测集合的下一项并设置位置。

如果有更多项，状态机会转入suspended状态，如果没有更多项，它转入并保持在after状态。图20-11演示了这个状态机。

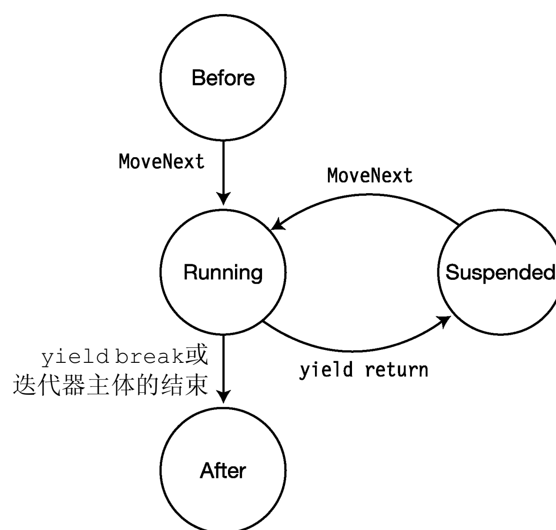



图20-11　迭代器状态机

# 第 21 章 介绍LINQ

**本章内容**

- ❑ 什么是LINQ
- ❑ LINQ提供程序
- ❑ 查询语法和方法语法
- ❑ 查询变量
- ❑ 查询表达式的结构
- ❑ 标准查询运算符
- ❑ LINQ to XML

## 21.1 什么是 LINQ

LINQ是集成到C#和Visual Basic.NET这些语言中用于提供查询数据能力的一个新特性。

在关系型数据库系统中，数据被组织放入规范化很好的表中，并且通过简单而又强大的语言SQL来进行访问。SQL可以和数据库中的任何数据配合使用，因为数据被放入表中，并遵从一些严格的规则。

然而，在程序中却与数据库相反，保存在类对象或结构中的数据差异很大。因此，没有通用的查询语言来从数据结构中获取数据。自C# 3.0引入LINQ以后，该语言便有了查询对象集合的能力。如下是LINQ的重要高级特性：

- ❑ LINQ（发音为link）代表语言集成查询（Language Integrated Query）。
- ❑ LINQ是.NET框架的扩展，它允许我们以数据库查询的方式查询数据集合。
- ❑ C# 3.0包含整合LINQ到语言中的一些扩展，允许我们从数据库、程序对象集合以及XML文档中查询数据。

如下代码演示了一个最简单的使用LINQ的示例。在这段代码中，被查询的数据源是简单的int数组。语句中查询的定义就是from和select关键词。尽管查询在语句中被定义，但是查询其实是在最后的foreach语句中进行的。

```
static void Main()
{
   int[] numbers = { 2, 12, 5, 15 };           //数据源
```

```
    IEnumerable<int> lowNums =                  // 定义并存储查询
                        from n in numbers
                        where n < 10
                        select n;

    foreach (var x in lowNums)                  // 执行查询
        Console.Write("{0}, ", x);
}
```

这段代码产生了如下的输出：

```
2, 5,
```

## 21.2 LINQ 提供程序

在之前的示例中，数据源是简单的int数组，它是程序在内存中的对象。然而，LINQ还可以和各种类型的数据源一起工作，比如SQL数据库、XML文档，等等。然而，对于每一种数据源类型，在其背后一定有根据该数据源类型实现LINQ查询的代码模块。这些代码模块被叫做LINQ提供程序（provider）。有关LINQ提供程序的要点如下：

- 微软提供常见数据源类型的一些LINQ提供程序，如图21-1所示。
- 我们可以使用任何支持LINQ的语言（在这里是C#）来查询有LINQ提供程序的数据源类型。
- 第三方在不断提供针对各种数据源类型的LINQ提供程序。

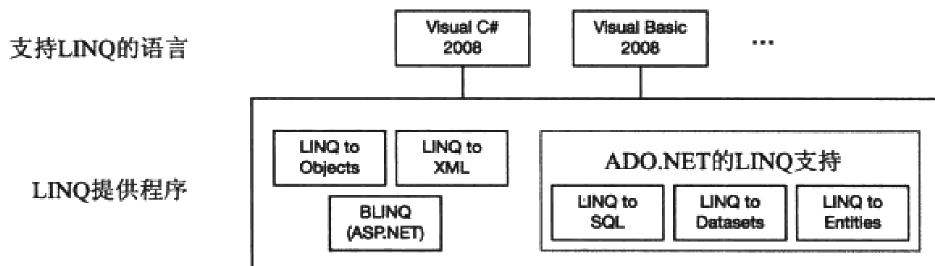



图21-1　LINQ的体系结构，支持LINQ的语言以及LINQ提供程序

有很多介绍LINQ的各种形式和细节的专著，毫无疑问，这些内容超过了本章讨论的范围。在本章中，我们会介绍LINQ以及解释它如何与程序对象（LINQ to Objcts）以及XML配合使用（LINQ to XML）。

### 匿名类型

在介绍LINQ查询特性的细节之前，让我们首先学习一下C# 3.0的允许我们创建无名类类型的特性。不足为奇，这被叫做*匿名类型*（anonymous type）。

在第5章中，我们介绍了对象初始化器，它允许我们在使用对象创建表达式时初始化新类实

例的字段和属性。只是提醒一下，这种形式的对象创建表达式由三部分组成：new关键词、类名或构造函数以及对象初始化器。对象初始化器在一组花括号内包含了逗号分隔的成员初始化列表。

创建匿名类型的变量使用相同的形式，但是没有列名和构造方法。如下的代码行演示了匿名类型对象创建的形式：

```
                 对象初始化列表
                       ↓
new { FieldProp = InitExpr, FieldProp = InitExpr, ...}
             ↑                    ↑
        成员初始化列表         成员初始化列表
```

如下代码给出了一个创建和使用匿名类型的实例。它创建了一个叫做student的变量，这是一个有三个string属性和一个int属性的匿名类型。注意，在WriteLine语句中，可以像访问具名类型的成员那样访问实例的成员。

```
static void Main( )
{
   var student = new {LName="Jones", FName="Mary", Age=19, Major="History"};
    ↑                                        ↑
  使用var                                对象初始化列表

   Console.WriteLine("{0} {1}, Age {2}, Major: {3}",
        student.FName, student.LName, student.Age, student.Major);
}
```

这段代码产生了如下的输出：

```
Mary Jones, Age 19, Major: History
```

需要了解的有关匿名类型的重要事项如下：

- 匿名类型只能和局部变量配合使用，不能用于类成员。
- 由于匿名类型没有名字，我们必须使用var关键词作为变量类型。

当编译器遇到匿名类型的对象初始化器时，它创建了一个有名字的新类类型。对于每一个成员初始化器，它推断其类型并在新的类中创建这个类型的私有变量，然后创建用于访问这个变量的读写属性。属性和成员初始化器具有相同的名字。匿名类型被构造后，编译器创建了这个类型的对象。

除了对象初始化器的赋值形式，匿名类型的对象初始化器还有其他两种允许的形式：简单标识符和成员访问表达式。这两种形式叫做*投影初始化器*（projection initializer）。下面的变量声明演示了所有的三种形式。第一个成员初始化器是赋值形式，第二个是标识符形式，第三个是成员访问表达式。

```
var student = new { Age = 19, Major, Other.Name };
```

例如，如下代码使用了所有的三种类型。注意，投影初始化器定义在匿名类型声明之前。Major

是一个局部变量，Name是Other类的静态字段。

```
class Other
{
   static public string Name = "Mary Jones";
}

class Program
{
   static void Main()
   {
      string Major = "History";
                              赋值形式           标识符
                              ↓                  ↓
      var student = new { Age = 19, Other.Name, Major};
                                        ↑
                                     成员访问

      Console.WriteLine("{0}, Age {1}, Major: {2}",
                 student.Name, student.Age, student.Major);
   }
}
```

这段代码产生了如下的输出：

```
Mary Jones, Age 19, Major: History
```

刚才演示的映射初始化器形式和这里给出的赋值形式的结果一样：

```
var student = new { Age = Age, Name = Other.Name, Major = Major};
```

尽管在代码中看不到匿名类型，对象浏览器却能看到。如果编译器遇到了另一个具有相同参数名、相同引用类型名和相同顺序的匿名类型，它会重用这个类型并直接创建新的实例，而不会创建新的匿名类型。

## 21.3 查询语法和方法语法

有两种形式的语法可供我们在写LINQ查询时使用——查询语法和方法语法。

- 查询语法（query syntax）是声明形式的，看上去和SQL语句很相似。查询语法使用查询表达式形式书写。
- 方法语法（method syntax）是命令形式的，它使用的是标准的方法调用。方法是一组叫做标准查询运算符的方法，本章稍后会介绍。
- 在一个查询中也可以组合两种形式。

微软推荐使用查询语法，因为它更易读，能更清晰地表明查询意图，因此也更不容易出错。然而，有一些运算符必须使用方法语法来书写。

**说明**　查询表达式使用的查询语法会被C#编译器转换为方法调用的形式。这两种形式在运行时性能上没有区别。

如下代码演示了所有的三种查询形式。对于方法语法的那部分代码，你可能觉得Where方法的参数看起来很奇怪，这是lambda表达式（第15章也介绍过），在本章后面还会介绍它在LINQ中的使用。

```
static void Main( )
{
   int[] numbers = { 2, 5, 28, 31, 17, 16, 42 };

   var numsQuery = from n in numbers                         // 查询语法
                   where n < 20
                   select n;

   var numsMethod = numbers.Where(x => x < 20);              // 方法语法

   int numsCount = (from n in numbers                        // 两种形式的组合
                    where n < 20
                    select n).Count();

   foreach (var x in numsQuery)
      Console.Write("{0}, ", x);
   Console.WriteLine();

   foreach (var x in numsMethod)
      Console.Write("{0}, ", x);
   Console.WriteLine();

   Console.WriteLine(numsCount);
}
```

这段代码产生了如下的输出：

```
2, 5, 17, 16,
2, 5, 17, 16,
4
```

## 21.4　查询变量

LINQ查询可以返回两种类型的结果——一个枚举①，它列出了满足查询参数的项列表；一个叫做标量（scalar）的单一值，它是满足查询条件的结果的某种摘要形式。

例如，第一行代码返回的是一个IEnumerable对象，它可以被用来枚举查询的结果。第二个

① 可枚举的一组数据，不是枚举类型。——译者注

语句执行查询并调用一个方法（Count）来返回从查询返回的项的总数，在本章后面我们会介绍诸如Count等返回标量的运算符。

```
int[] numbers = { 2, 5, 28 };

IEnumerable<int> lowNums = from n in numbers        // 返回枚举数
                           where n < 20
                           select n;

int numsCount           = (from n in numbers        // 返回一个整数
                           where n < 20
                           select n).Count();
```

等号左边的变量叫做查询变量。尽管在前面的语句中查询变量的类型被显式定义了，我们还是可以使用var关键词替代变量名称来让编译器自行推断查询变量的类型。

理解查询变量的内容很重要。在执行前面的代码后，lowNums查询变量不会包含查询的结果，它包含的是IEnumerable<int>类型的对象。查询变量numCount包含的是真实的整数值，它只能通过真实运行查询后获得。

区别在于查询执行的时间，可以总结如下：

- 如果查询表达式返回枚举，查询一直到处理枚举时才会被执行。如果枚举被处理多次，查询就会被执行多次。
- 如果查询表达式返回标量，查询立即执行，并且把结果保存在查询变量中。

图21-1演示了可枚举查询的情况。变量lowNums包含指向可从中枚举查询结果数组对象的引用。

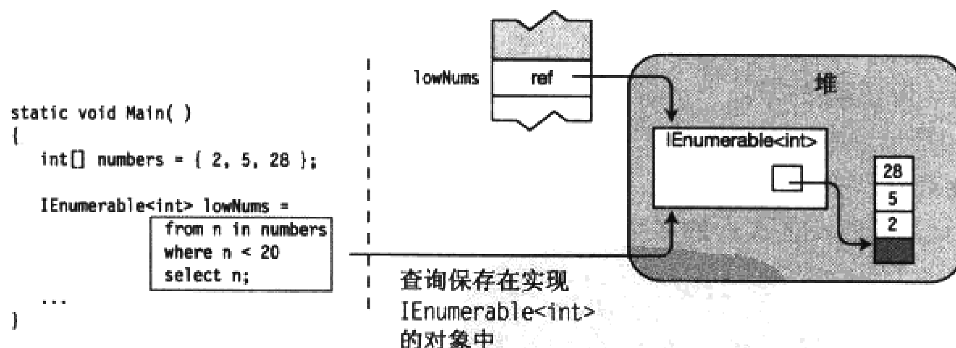



图21-2　编译器创建一个实现IEnumerable<int>的对象并把查询保存于其中

## 21.5　查询表达式的结构

如图21-3所示，查询表达式由查询体后的from子句组成。有关查询表达式需要了解的一些重要事项如下：

- 子句必须按照一定的顺序出现。
  - from子句和select…group子句这两部分是必须的。
  - 其他子句是可选的。

- 在LINQ查询表达式中，select子句在表达式最后。这与SQL的SELECT语句在查询的开始处不一样。C#这么做的原因之一是让Visual Studio智能感应能在我们输入代码时给我们更多选项。
- 可以有任意多的from…let…where子句，如图21-3所示。

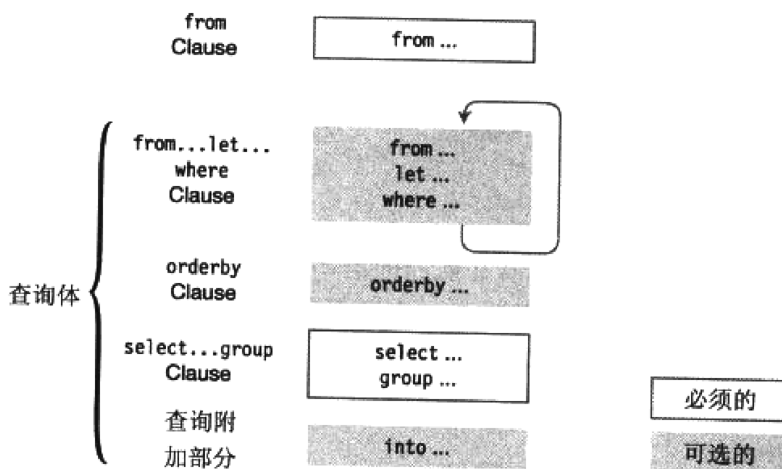



图21-3　查询语句的结构由from子句后面跟查询体开始

### 21.5.1　from子句

from子句指定了要作为数据源使用的数据集合。它也引入了迭代变量。有关from子句的要点如下所示。

- 迭代变量有序表示数据源的每一个元素。
- from子句的语法如下：
  - Type是集合中元素的类型。这是可选的，因为编译器可以从集合来推断类型。
  - Item是迭代变量的名字。
  - Items是要查询的集合的名字。集合必须是可枚举的，见第13章。

```
        迭代变量声明
             ↓
From Type Item in Items
```

如下的代码给出了用于查询四个int数组的查询表达式。迭代变量item会表现数组中的每一个元素，并且会被之后的where和select子句选择或丢弃。这段代码没有指明迭代变量的可选类型（int）。

```
int[] arr1 = {10, 11, 12, 13};
                   迭代变量
                      ↓
var query = from item in arr1
            where item < 13        ← 使用迭代变量
            select item;           ← 使用迭代变量
```

```
foreach( var item in query )
   Console.Write("{0}, ", item );
```

这段代码产生了如下的输出：

```
10, 11, 12,
```

图21-4演示了from子句的语法。类型说明符是可选的，因为可以由编译器推断。可以有任何多个join子句。

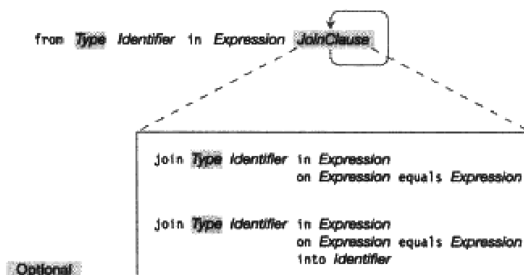



图21-4 from子句语法

尽管LINQ的from子句和foreach语句非常相似，但是主要的不同点如下：

- foreach语句在遇到代码时就执行其主体，而from子句什么也不执行。它创建一个用于保存查询变量的可枚举对象。查询本身会在之后的代码中被执行或不被执行。
- foreach语句明确指定集合中的项需要按照次序，从第一个到最后一个。而from子句只是声明性地规定了必须考虑集合中的每一个项，不规定其顺序。

### 21.5.2 join 子句

LINQ中的join子句和SQL中的JOIN子句很相似。如果你熟悉SQL中的联结，从概念上来说，LINQ中的联结对你来说应该不是新鲜事。不同的是，我们现在不但可以在数据库的表上这么做，而且还可以在集合对象上进行这个操作。如果你不熟悉联结或需要重新了解它，那么下面内容可能会帮你理清思路。

需要先了解的有关联结的语法如下：

- 联结操作接受两个集合然后创建一个临时的对象集合，每一个对象包含原始集合对象中的所有字段。
- 使用连接来结合两个或更多集合中的数据。

联结的语法如下，它指定了用于与之前子句中集合进行连接的第二个集合。

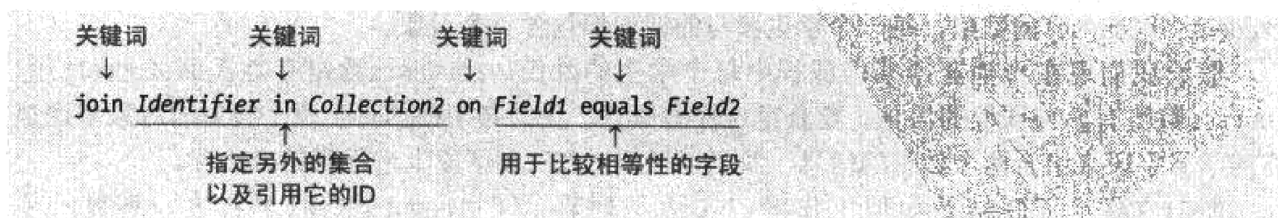


图21-5演示join子句的语法。

```
join Type Identifier in Expression
                     on Expression equals Expression

join Type Identifier in Expression
                     on Expression equals Expression
                     into Identifier
```

图21-5　联结子句的语法

如下具有注解的语句给出了一个join子句的示例：

```
              第一个集合和ID
                   ↓                 第一个集合的项   第二个集合的项
var query = from s in students                 ↓               ↓
            join c in studentsInCourses on s.StID equals c.StID
                        ↑                           ↑
                  第二个集合和ID                 比较的字段
```

## 21.5.3　什么是联结

LINQ中的联结接受两个集合然后创建一个临时的对象集合，每一个元素包含两个原始集合中的原始成员。

例如，如下的代码声明了两个类：Student和CourseStudent。

- Student类型的对象包含了学生的姓氏和学号。
- CourseSrudent类型的对象表示参与课程的学生，它包含课程名以及学生的ID。

```
public class Student
{
   public int     StID;
   public string LastName;
}

public class CourseStudent
{
   public string CourseName;
   public int     StID;
}
```

图21-6演示了程序中的情况，在这里有三个学生和三门课程，学生参加不同的课程。程序有一个Student对象构成的叫做students的数组，以及一个由CourseStudent对象构成的叫做studentsInCourses的数组，每一个学生参与的课程都包含一个对象。

假设我们现在希望获得某门课程中每个学生的姓氏。students数组有姓氏而studentsIn-Courses数组有参与课程的信息。要获得这样的信息，我们必须根据对于两个类型的对象来说都有的学号字段来组合两个数组的信息。我们可以通过在StID字段上进行联结来实现。

图21-7演示了联结是如何工作的。左边一列显示了students数组,而右边一列显示了

studentsInCourses对象。如果我们拿第一个学生的记录并把它的ID和每一个studentsInCourses对象中的学生ID进行比较，我们可以找到两条匹配的记录，中间列的顶部就是。然后，我们对其他两个学生也执行相同的操作，会发现第二个学生选了一门课程而第三个学生选了两门课程。

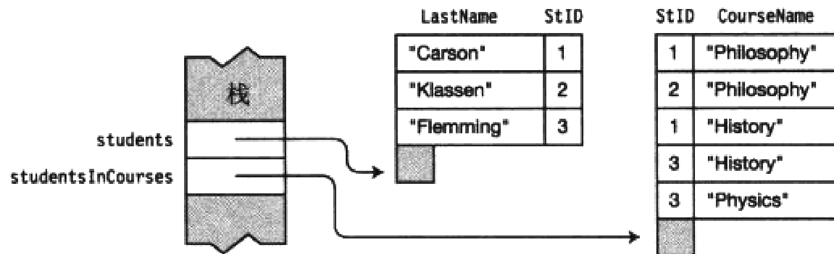



图21-6 学生参与的各种课程

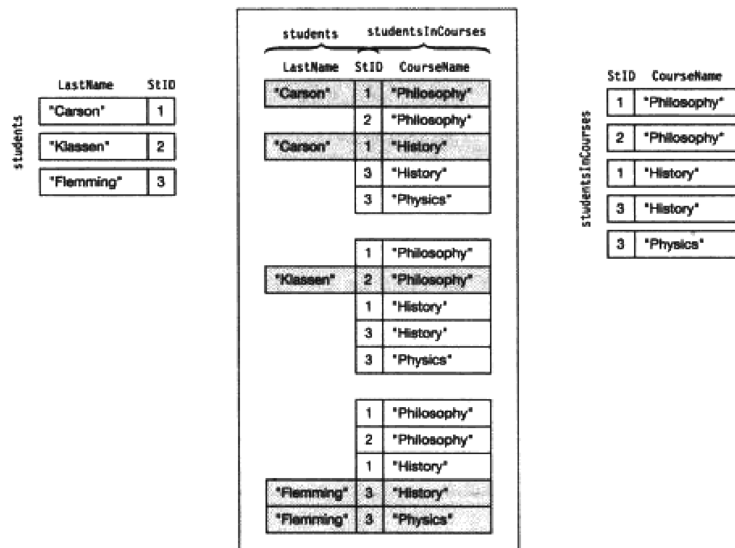



图21-7 两个数组的对象以及它们在StID上联结的结果

中间一列5个灰色的对象表示两个数组在StID字段上进行联结。每一个对象包含了三个字段：Student类的LastName字段，CourseStudent类的CourseName字段以及两个类共有的StID字段。

如下的代码把整个示例放在了一起，查询找出了所有选择历史课的学生的姓氏。

```
class Program
{
   public class Student {
      public int    StID;
      public string LastName;
   }

   public class CourseStudent {
      public string CourseName;
      public int    StID;
   }
```

```
    static CourseStudent[] studentsInCourses = new CourseStudent[] {
        new CourseStudent { CourseName = "Art",        StID = 1 },
        new CourseStudent { CourseName = "Art",        StID = 2 },
        new CourseStudent { CourseName = "History",    StID = 1 },
        new CourseStudent { CourseName = "History",    StID = 3 },
        new CourseStudent { CourseName = "Physics",    StID = 3 },
     };

    static Student[] students = new Student[] {
        new Student { StID = 1, LastName = "Carson"   },
        new Student { StID = 2, LastName = "Klassen"  },
        new Student { StID = 3, LastName = "Flemming" },
     };

    static void Main( )
    {
      // 查找所有选择了历史课的学生的姓氏
      var query = from s in students
                  join c in studentsInCourses on s.StID equals c.StID
                  where c.CourseName == "History"
                  select s.LastName;

      // 显式所有选择了历史课的学生的名字
      foreach (var q in query)
         Console.WriteLine("Student taking History:  {0}", q);
    }
 }
```

这段代码产生了如下的输出：

```
Student taking History:  Carson
Student taking History:  Flemming
```

## 21.5.4　查询主体中的from…let…where片段

可选的from…let…where部分是查询主体的第一部分，可以由任意数量的三个子句来组合——from子句、let子句和where子句。图21-8总结了这些子句的语法。

**1. from子句**

我们看到查询表达式从必须的from子句开始，后面跟的是查询主体。主体本身可以从任何数量的其他from子句开始，每一个from子句都指定了一个额外的数据集合并引入了要在之后运算的迭代变量，所有from子句的语法和含义都是一样的。

如下代码演示了这种用法的一个示例。

- 第一个from子句是查询表达式必须的子句。
- 第二个from子句是第一个子句的查询主体。

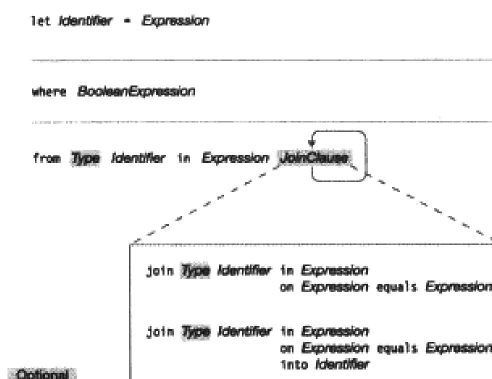



图21-8 from…let…where子句的语法

- select子句创建了一个匿名类型的对象。本章的前面介绍过匿名对象，之后还会介绍如何在查询表达式中使用它。

```
static void Main()
{
   var groupA = new[] { 3, 4, 5, 6 };
   var groupB = new[] { 6, 7, 8, 9 };

   var someInts = from a in groupA                ← 必须的第一个from子句
                  from b in groupB                ← 查询主体的第一个子句
                  where a > 4 && b <= 8
                  select new {a, b, sum = a + b}; ← 匿名类型对象

   foreach (var a in someInts)
      Console.WriteLine(a);
}
```

这段代码产生了如下的输出：

```
{ a = 5, b = 6, sum = 11 }
{ a = 5, b = 7, sum = 12 }
{ a = 5, b = 8, sum = 13 }
{ a = 6, b = 6, sum = 12 }
{ a = 6, b = 7, sum = 13 }
{ a = 6, b = 8, sum = 14 }
```

### 2. let子句

let子句接受一个表达式的运算并且把它赋值给一个需要在其他运算中使用的标识符。let子句的语法如下：

```
let Identifier = Expression
```

例如，如下代码中的查询表达式将数组groupA中的每一个成员与数组groupB中的每一个成员进行配对。where子句去除两个数组的整数相加不等于12的组合。

```
static void Main()
{
   var groupA = new[] { 3, 4, 5, 6 };
   var groupB = new[] { 6, 7, 8, 9 };

   var someInts = from a in groupA
                  from b in groupB
                  let sum = a + b         ← 在新的变量中保存结果
                  where sum == 12
                  select new {a, b, sum};

   foreach (var a in someInts)
      Console.WriteLine(a);
}
```

这段代码产生了如下的输出：

```
{ a = 3, b = 9, sum = 12 }
{ a = 4, b = 8, sum = 12 }
{ a = 5, b = 7, sum = 12 }
{ a = 6, b = 6, sum = 12 }
```

### 3. where子句

where子句根据之后的运算来去除不符合指定条件的项。where子句的语法如下：

```
where BooleanExpression
```

有关where需要了解的重要事项如下：

- 只要是在from…let…where部分中，查询表达式可以有任何多个where子句。
- 一个项必须满足where子句才能避免在之后被过滤。

如下代码给出了一个包含两个where子句的查询表达式的示例。where子句去除了两个数组中整数相加没有大于或等于11的组合，以及groupA中元素不等于值4的项。每一组选择的元素必定是满足两个where子句条件的。

```
static void Main()
{
   var groupA = new[] { 3, 4, 5, 6 };
   var groupB = new[] { 6, 7, 8, 9 };

   var someInts = from int a in groupA
                  from int b in groupB
                  let sum = a + b
                  where sum >= 11          ← 条件1
                  where a == 4             ← 条件2
                  select new {a, b, sum};

   foreach (var a in someInts)
```

```
        Console.WriteLine(a);
    }
```

这段代码产生了如下的输出：

```
{ a = 4, b = 7, sum = 11 }
{ a = 4, b = 8, sum = 12 }
{ a = 4, b = 9, sum = 13 }
```

### 21.5.5 orderby 子句

orderby子句接受一个表达式并根据表达式依次返回结果项。

orderby子句的语法如图21-9所示。可选的ascending和descending关键词设置了排序方向。表达式通常是项的一个字段。

- orderby子句的默认排序是升序。然而，我们可以使用ascending和descending关键词显式地设置元素的排序为升序或降序。
- 可以有任意多个子句，它们必须使用逗号分隔。




图21-9 orderby子句的语法

如下代码给出了一个按照学生年龄排序学生记录的示例。注意，学生数据数组保存在一个匿名类型数组中。

```
static void Main( ) {
   var students = new []        // 匿名类型的对象数组
   {
      new { LName="Jones",   FName="Mary",   Age=19, Major="History" },
      new { LName="Smith",   FName="Bob",    Age=20, Major="CompSci" },
      new { LName="Fleming", FName="Carol",  Age=21, Major="History" }
   };

   var query = from student in students
               orderby student.Age        ← 根据Age排序
               select student;

   foreach (var s in query) {
      Console.WriteLine("{0}, {1}:  {2} - {3}",
               s.LName, s.FName, s.Age, s.Major);
   }
}
```

这段代码产生了如下的输出：

```
Jones, Mary:  19 - History
Smith, Bob:  20 - CompSci
Fleming, Carol:  21 - History
```

### 21.5.6　select…group 子句

有两种类型的子句组成select…group部分——select子句以及group…by子句。select…group部分之前的子句指定了数据源和要选择的对象，select…group部分的功能如下所示。

- select子句指定所选对象的哪部分应该被select。它可以指定下面的任意一项：
  - 整个数据项。
  - 数据项的一个字段。
  - 数据项中几个字段组成的新对象（或类似其他值）。
- group…by子句是可选的，用来指定选择的项如何被分组。我们会在本章稍后介绍group…by子句。

select…group子句的语法如图 21-10所示。

select *Expression*

group *Expression1* by *Expression2*

图21-10　select…group子句语法

如下代码给出了一个使用select子句选择整个数据项的示例。首先，我们创建了一个匿名类型对象的数组；然后，查询表达式使用select语句来选择数组中的每一项。

```
using System;
using System.Linq;
class Program {
   static void Main() {
      var students = new[]        // 匿名类型的对象数组
      {
         new { LName="Jones",   FName="Mary",  Age=19, Major="History" },
         new { LName="Smith",   FName="Bob",   Age=20, Major="CompSci" },
         new { LName="Fleming", FName="Carol", Age=21, Major="History" }
      };

      var query = from s in students
                  select s;

      foreach (var q in query)
         Console.WriteLine("{0}, {1}: Age {2}, {3}",
                           q.LName, q.FName, q.Age, q.Major);
   }
}
```

这段代码产生了如下的输出：

```
Jones, Mary: Age 19, History
Smith, Bob: Age 20, CompSci
Fleming, Carol: Age 21, History
```

我们也可以使用select子句来选择对象的某些字段。例如，如下代码中的select子句只select了学生名。

```
var query = from s in students
            select s.LName;
```

```
    foreach (var q in query)
        Console.WriteLine(q);
```

如果我们用这两条语句替换原来完整示例中对应的两条语句，程序会产生如下的输出：

```
Jones
Smith
Fleming
```

## 21.5.7 查询中的匿名类型

查询结果可以由原始集合的项、原始集合中项的一些字段或匿名类型组成。

我们可以通过在select子句中把希望在类型中包括的字段以逗号分隔，并以花括号进行包围来创建匿名类型。例如，要让之前部分的代码只选择学生姓名和主修课，我们可以使用如下的语法：

```
select new { s.LastName, s.FirstName, s.Major };
                         ↑
                      匿名类型
```

例如，如下代码在select子句中创建一个匿名类型并且在之后使用WriteLine语句。

```
using System;
using System.Linq;

class Program {
   static void Main()
   {
      var students = new[]
      {
         new { LName="Jones",   FName="Mary",   Age=19, Major="History" },
         new { LName="Smith",   FName="Bob",    Age=20, Major="CompSci" },
         new { LName="Fleming", FName="Carol",  Age=21, Major="History" }
      };

      var query = from s in students
                  select new { s.LName, s.FName, s.Major };
                                      ↑
                                  创建匿名类型

      foreach (var q in query)
         Console.WriteLine("{0} {1} -- {2}",
                       q.FName, q.LName, q.Major );
                                ↑
   }                      匿名类型的访问字段
}
```

这段代码产生了如下的输出：

```
Mary Jones -- History
Bob Smith -- CompSci
Carol Fleming -- History
```

### 21.5.8　group 子句

group子句把select的对象根据一些标准进行分组。例如，有了之前示例的学生数组，程序可以根据它们的主修课程进行分组。

有关group子句需要了解的重要事项如下：

- 如果项包含在查询的结果中，它们就可以根据某个字段的值进行分组。作为分组依据的项叫做键（key）。
- 和select子句不同，group子句不从原始的数据源中返回可枚举项的可枚举类型，而是返回可以枚举已经形成的项的分组的可枚举类型。
- 分组本身是可枚举类型，它们可以枚举实际的项。

group子句语法的一个示例如下：

```
group student by student.Major;
  ↑           ↑
关键字        关键字
```

例如，如下代码根据学生的主修课程进行分组：

```
static void Main( )
{
   var students = new[]
   {
      new { LName="Jones",   FName="Mary",   Age=19, Major="History" },
      new { LName="Smith",   FName="Bob",    Age=20, Major="CompSci" },
      new { LName="Fleming", FName="Carol",  Age=21, Major="History" }
   };

   var query = from student in students
               group student by student.Major;

   foreach (var s in query)            // 枚举分组
   {
      Console.WriteLine("{0}", s.Key);
                                ↑
                              分组键

      foreach (var t in s)             // 枚举分组中的项
         Console.WriteLine("      {0}, {1}", t.LName, t.FName);
   }
}
```

这段代码产生了如下的输出：

```
History
    Jones, Mary
    Fleming, Carol
CompSci
    Smith, Bob
```

图21-11演示了从查询表达式返回对象并保存于查询变量中。

- 从查询表达式返回的对象是从查询中枚举分组结果的可枚举类型。
- 每一个分组由一个叫做键的字段区分。
- 每一个分组本身是可枚举类型并且可以枚举它的项。

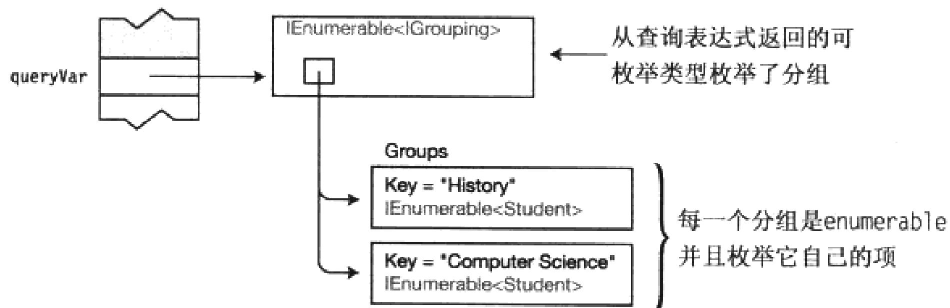



图21-11 group子句返回集合的集合对象而不是一个集合对象

## 21.5.9 查询延续

查询延续子句可以接受查询的一部分结果并赋予一个名字，从而可以在查询的另一部分中使用。查询延续的语法如图21-12所示。

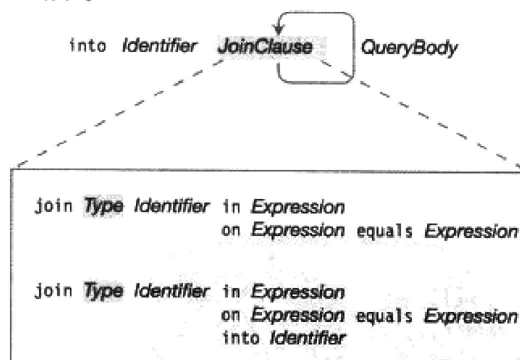



图21-12 查询延续子句的语法

例如，如下查询连接了groupA和groupB，并且命名为groupAandB，然后从groupA和groupB中进行一个简单的select。

```
static void Main()
{
   var groupA = new[] { 3, 4, 5, 6 };
   var groupB = new[] { 4, 5, 6, 7 };
```

```
    var someInts = from a in groupA
                   join b in groupB on a equals b
                   into groupAandB                ← 查询继续
                   from c in groupAandB
                   select c;

    foreach (var a in someInts)
       Console.Write("{0}  ", a);
}
```

这段代码产生了如下的输出：

```
4  5  6
```

## 21.6　标准查询运算符

标准查询运算符由一系列叫做API的方法组成，它能让我们查询任何.NET数组或集合。有关标准查询运算符的重要特性如下：

- 被查询的集合对象叫做序列，它必须实现IEnumerable<T>接口，T是类型。
- 标准查询运算符使用方法语法。
- 一些运算符返回IEnumerable对象（或其他序列），而其他的一些运算符返回标量。返回标量的运算符立即执行，而返回替代可枚举类型对象的值会被延迟迭代。

例如，如下代码演示了Sum和Count运算符的使用，并且返回了int。代码需要注意的地方如下所示：

- 用作方法的运算符直接作用于序列对象，在这里就是numbers数组。
- 返回类型不是IEnumerable对象，而是int。

```
class Program
{
   static int[] numbers = new int[] {2, 4, 6};

   static void Main( )
   {
      int total    = numbers.Sum();
      int howMany = numbers.Count();
       ↑             ↑          ↑
      标量          序列对象   运算符

      Console.WriteLine("Total: {0}, Count: {1}", total, howMany);
   }
}
```

这段代码产生了如下的输出：

```
Total: 12, Count: 3
```

47个标准查询运算符可以分成14个不同的分类，表21-1列出了这些分类。

**表21-1 标准查询运算符的分类**

| 名字 | 运算符数量 | 描述 |
|---|---|---|
| 限制 | 1 | 根据选择的标准返回序列对象的子集 |
| 映射 | 2 | 选择最终返回序列对象的哪些部分 |
| 分隔 | 4 | 从序列跳过或返回对象 |
| 联结 | 2 | 根据一些标准，返回联结两个序列的IEnumerable对象 |
| 合成 | 1 | 从两个单独的序列产生一个序列 |
| 排序 | 2 | 根据提供的标准排序序列 |
| 分组 | 1 | 根据提供的标准分组序列 |
| 设置 | 4 | 在序列上进行设置操作 |
| 转换 | 7 | 把序列转化为数组、列表以及字典等各种形式 |
| 判等 | 1 | 判断两个序列是否相等 |
| 元素 | 9 | 返回序列中的特定元素 |
| 生成 | 3 | 生成序列 |
| 量化 | 3 | 返回一个指定某个谓词对于序列是否为true的布尔值 |
| 聚合 | 7 | 返回一个表示序列特性的单一值 |

### 21.6.1 查询表达式和标准查询运算符

如本章开始所讲，每一个查询表达式还可以使用带有标准查询运算符的方法语法来编写。标准查询运算符是进行查询的一组方法。编译器把每一个查询表达式翻译成标准查询运算符形式。

很明显，由于所有查询表达式被翻译成标准查询运算符——运算符可以执行由查询表达式完成的任何操作，而且运算符还有查询表达式形式所不能提供的附加功能。例如，在之前示例中使用的Sum和Count运算符，只能使用方法语法来表达。

然而，查询表达式和方法语法这两种表达式也可以组合。例如，如下代码演示了使用了Count运算符的查询表达式。注意，在该代码中，查询表达式是在圆括号内的一部分，在它之后跟一个点和方法的名字。

```
static void Main()
{
   var numbers = new int[] { 2, 6, 4, 8, 10 };

   int howMany = (from n in numbers
                  where n < 7
                  select n).Count();
                     ↑         ↑
                  查询表达式   运算符

   Console.WriteLine("Count: {0}", howMany);
}
```

这段代码产生了如下的输出：

```
Count: 3
```

## 21.6.2 标准查询运算符的签名

System.Linq.Enumerable类声明了标准查询运算符方法。然而，这些方法不仅仅是一些方法，它们是扩展了IEnumerable<T>泛型的扩展方法。

第7章和第19章中介绍了扩展方法，要记住的最重要的事情就是它们是公共的静态方法，尽管定义在一个类中，但目的是为另一个类增加功能——所列的第一个形参。参数前必须有关键词this。

例如，如下是三个运算符的签名：Count、First和Where。乍看上去运算符的签名很吓人。注意下面有关签名的事项：

- 由于运算符是泛型方法，它们有与它们的名字相关联的泛型参数（T）。
- 由于运算符是扩展IEnumerable的扩展方法，它们必须满足下面的语法条件：
  - 它们必须声明为public和static。
  - 它们必须在第一个参数前有this扩展指示器。
  - 它们必须把IEnumerable<T>作为第一个参数。

```
始终是public和static           名字和泛型参数    第一个参数
        ↓                            ↓                ↓
public static      int         Count<T>( this IEnumerable<T> source );
public static       T          First<T>( this IEnumerable<T> source );
public static IEnumerable<T>   Where<T>( this IEnumerable<T> source, ... );
                    ↑                     ↑
                 返回类型              扩展指示器
```

例如，如下代码演示了使用Count和First运算符。两个运算符都接受一个参数——IEnuerable<T>对象的引用。

- Count运算符返回序列中所有元素数量的单一值。
- First运算符返回序列中的第一个元素。

前两次代码中使用的运算符都是直接调用的，和普通方法差不多，传入数组的名字作为第一个参数。然而，之后的两行代码使用扩展方法语法来调用，就好像它们是数组（enumerable）的方法成员一样。注意，在这里没有指定参数，而是数组名称从参数列表中移到了方法名称之前，用起来就好像它包含了方法的声明一样。

直接语法调用和扩展语法调用是完全相等的——除了语法不同之外。

```
using System.Linq;
   ...
static void Main( )
{
```

```
    int[] intArray = new int[] { 3, 4, 5, 6, 7, 9 };
                                         数组作为参数
                                              ↓
    var count1     = Enumerable.Count(intArray);   // 直接调用
    var firstNum1 = Enumerable.First(intArray);    // 直接调用

    var count2     = intArray.Count();             // 扩展方法调用
    var firstNum2 = intArray.First();              // 扩展方法调用
                          ↑
                  数组作为被扩展的对象

    Console.WriteLine("Count: {0}, FirstNumber: {1}", count1, firstNum1);
    Console.WriteLine("Count: {0}, FirstNumber: {1}", count2, firstNum2);
}
```

这段代码产生了如下的输出：

```
Count: 6, FirstNumber: 3
Count: 6, FirstNumber: 3
```

## 21.6.3 委托作为参数

在之前的部分内容中我们已经看到了，每一个运算符的第一个参数是IEnumerable<T>对象的引用，之后的参数可以是任何类型。很多运算符接受泛型委托作为参数(泛型委托在第19章中解释过)。有关把泛型委托作为参数需要了解的最重要事项：泛型委托用于给运算符提供用户自定义的代码。

为了解释该事项，我们首先从一个演示Count运算符的几种使用方式的示例开始。Count运算符被重载并且有两种形式。第一种形式在之前的示例中用过，如下所示，它有一个参数：

```
public static int Count<T>(this IEnumerable<T> source);
```

和所有扩展方法差不多，我们可以使用标准静态方法形式或在一个扩展的类上使用实例方法的形式，如下面两行代码所示：

```
var count1 = Linq.Enumerable.Count(intArray);       // 静态方法形式

var count2 = intArray.Count();                      // 实例方法形式
```

在这两个实例中，查询统计给定的整数数组中的int的总数。然而，假设我们希望看看数组中奇数元素的总数，我们必须指定Count方法，检测整数是否是奇数的方法才能实现。

我们可能需要使用Count方法的第二种形式才能实现，如下所示。它有一个泛型委托作为其第二个参数。调用时，我们必须提供一个接受单个T类型的输入参数并返回布尔值的委托对象。委托代码的返回值必须指定元素是否包含在总数中。

```
public static int Count<T>(this IEnumerable<T> source,
                            Func<T, bool> predicate );
                                   ↑
                                泛型委托
```

例如，如下代码使用了第二种形式的Count运算符并指导它来包含那些奇数值。它通过提供一个lambda表达式来实现这个表达式在输入值是奇数时返回true，否则返回false。（lambda表达式在第15章中介绍过。）对于集合的每次遍历，Count调用这个方法（由lambda表达式表示）并把输入作为当前值。如果输入的是奇数，方法返回true，Count会把这个元素包含在总数中。

```
static void Main()
{
   int[] intArray = new int[] { 3, 4, 5, 6, 7, 9 };

   var countOdd = intArray.Count(n => n % 2 == 1);
                                      ↑
                              寻找奇数的Lambda表达式
   Console.WriteLine("Count of odd numbers: {0}", countOdd);
}
```

这段代码产生了如下的输出：

```
Count of odd numbers: 4
```

### 21.6.4　LINQ预定义的委托类型

和前面示例中的Count运算符差不多，很多LINQ运算符需要我们提供代码来指示运算符如何执行它的操作。我们通过把委托对象作为参数来实现。

在15章中我们把委托对象认为是一个包含指定参数和返回值的方法或方法列表的对象。当委托被调用时，包含它的方法会被连续调用。

LINQ定义了一套5种泛型委托类型与标准查询运算符一起使用。它们就是Func委托。

- 我们用作实参的委托对象必须是这5种类型或5种形式之一。
- TR代表返回值，并且总是在类型参数列表中的最后一个。

在这里列出了5个泛型委托对象。第一种形式没有接受方法参数，返回一个object作为返回值。第二个接受单个方法参数并且返回一个值，依此推类。

```
public delegate TR Func<TR>                ( );
public delegate TR Func<T1, TR >           ( T1 a1 );
public delegate TR Func<T1, T2, TR >       ( T1 a1, T2 a2 );
public delegate TR Func<T1, T2, T3, TR>    ( T1 a1, T2 a2, T3 a3 );
public delegate TR Func<T1, T2, T3, T4, TR>( T1 a1, T2 a2, T3 a3, T4 a4 );
                ↑            ↑                           ↑
             返回类型      类型参数                    方法参数
```

知道了这些，如果我们再来看一下Count的声明。我们可以发现第二个参数必须是委托对象，它接受单个T类型的参数作为方法参数并且返回一个bool类型的值。

```
public static int Count<T>(this IEnumerable<T> source,
                                         Func<T, bool> predicate );
                                              ↑    ↑
                                        参数类型  返回类型
```

产生一个布尔值的参数委托叫做谓词。

### 21.6.5 使用委托参数的示例

既然我们已经对Count签名以及LINQ中泛型委托参数有了更深入的理解，我们就可以更好地理解整个示例了。

如下代码先声明了IsOdd方法，它接受单个int类型的参数，并且返回表示输入参数是否是奇数的bool值。Main方法做了如下的事情：

- 它声明了int数组作为数据源。
- 它创建了一个类型为Func<int,bool>名称为MyDel的委托对象，并且使用IsOdd方法来初始化委托对象。注意，我们不需要声明Func代理类型，因为LINQ已经预定义了。
- 它使用委托对象调用Count。

```
class Program
{
   static bool IsOdd(int x)     // 委托对象使用的方法
   {
      return x % 2 == 1;        // 如果x是奇数，返回true
   }

   static void Main()
   {
      int[] intArray = new int[] { 3, 4, 5, 6, 7, 9 };

      Func<int, bool> myDel = new Func<int, bool>(IsOdd); // 委托对象
      var countOdd = intArray.Count(myDel);               // 使用委托

      Console.WriteLine("Count of odd numbers: {0}", countOdd);
   }
}
```

这段代码产生了如下的输出：

```
Count of odd numbers: 4
```

### 21.6.6 使用 Lambda 表达式参数的示例

之前的示例使用独立的方法和委托来把代码附加到运算符上。这需要声明方法和委托对象，然后把委托对象传递给运算符。如果下面的条件有任意一个是成立的，这种方式是不错的方案：

- 如果方法还需要在程序的其他地方被调用，而不仅仅是用来初始化委托对象。
- 如果函数体中的代码语句多余一条。

然而，如果这两个条件都不成立，我们可能希望使用更简洁和更局部化的方法来给运算符提供代码，那就是使用第15章中介绍的lambda表达式。

我们可以使用lambda表达式来修改之前的示例。首先，删除整个IsOdd方法，然后把等价的

lambda表达式直接替换委托对象的声明。新的代码更短更简洁，如下所示：

```
class Program
{
   static void Main()
   {
      int[] intArray = new int[] { 3, 4, 5, 6, 7, 9 };
                                    Lambda表达式
                                    ↓
      var countOdd = intArray.Count( x => x % 2 == 1 );

      Console.WriteLine("Count of odd numbers: {0}", countOdd);
   }
}
```

和之前的示例一样，这段代码产生了如下的输出：

```
Count of odd numbers: 4
```

如下所示，我们也可以使用匿名方法来替代lambda表达式。然而，这种方式比较累赘，而且lambda表达式在语义上与匿名方法是完全等价的，而且更简洁，因此没有理由再去使用匿名方法了。

```
class Program
{
   static void Main( )
   {
      int[] intArray = new int[] { 3, 4, 5, 6, 7, 9 };
                                   匿名方法
                                     ↓
      Func<int, bool> myDel = delegate(int x)
                              {
                                 return x % 2 == 1;
                              };
      var countOdd = intArray.Count(myDel);

      Console.WriteLine("Count of odd numbers: {0}", countOdd);
   }
}
```

## 21.7　LINQ to XML

在之前的几年里，XML已经变成一个保存和交换数据的重要方法。C# 3.0语言增加了一些特性来使我们与XML交互的工作比之前的XPath和XSLT等类似的方式更简单。如果你熟悉这些方式，可能会很愿意听到LINQ to XML在很多方面简化了XML的创建、遍历和操作，包括如下方面：

- 我们可以使用单一语句自顶向下创建XML树。

- 我们可以不使用包含树的XML文档在内存中操作XML。
- 我们可以不使用Text子节点来创建和操作字符串节点。

尽管我不会完整介绍XML，在介绍C# 3.0引入的XML操作特性之前，我首先会粗略介绍一下XML。

### 21.7.1 标记语言

标记语言（markup language）是文档中的一组标签，它提供有关文档的信息。也就是说，标记标签不是文档的数据——它们包含有关数据的数据。有关数据的数据称为元数据。

标记语言是被定义的一组标签，旨在传递有关文档内容的特定类型的元数据。例如，HTML是众所周知的标记语言。标签中的元数据包含了Web页面如何在浏览器中呈现以及如何使用超链接在页面中导航的信息。

大多数标记语言包含一组预定义的标签，而XML只包含少量预定义的标签，其他都由程序员来定义，来表示特定文档类型需要的任何元数据。只要数据的读者和编写者都知道标签的含义，标签就可以包含任何设计者希望的有用信息。

### 21.7.2 XML 基础

XML文档中的数据包含了一个XML树，它主要由嵌套元素组成。

元素是XML树的基本要素。每一个元素都有名字并且包含数据，一些元素还可以包含其他被嵌套的元素。元素由开始和关闭标签进行划分。任何元素包含的数据都必须介于开始和关闭标签之间。

- 开始标签从一个小于号开始，后面跟元素名，紧接着是可选的任何特性，最后是大于号。

```
<PhoneNumber>
```

- 关闭标签从一个小于号开始，后面是斜杠，然后是元素名和大于号。

```
<PhoneNumber>
```

- 没有内容的元素可以直接由单个标签表示，从小于号开始，后面是元素名和斜线，并使用大于号结束。

```
<PhoneNumber />
```

如下XML片段演示了一个叫做EmployeeName的元素，后面是空元素PhoneNumber。

```
<EmployeeName>Sally Jones</EmployeeName>
      ↑            ↑            ↑
   开始标签        内容        关闭标签
<PhoneNumber />      ← 没有内容的元素
```

其他需要了解的有关XML的重要事项如下：

- XML文档必须有一个根元素来包含所有其他元素。
- XML标签必须合理嵌套。
- 与HTML标签不同，XML标签是区分大小写的。
- XML特性是名字/值的配对，它包含了元素的额外元数据。特性的值部分必须包含在引号内，可以是单引号也可以是双引号。
- XML文档中的空格是有效的。这与把空格作为单个空格输出的HTML不同。

下面的XML文档是包含两个职工信息的XML示例。这个XML树非常简单，以清晰显示这些元素。需要了解的有关这个XML树的重要事项如下：

- 树包含了一个Employees类型的根节点，它包含了两个Employee类型的子节点。
- 每一个Employee节点包含了包含职工姓名和电话的节点。

```
<Employees>
   <Employee>
      <Name>Bob Smith</Name>
      <PhoneNumber>408-555-1000</PhoneNumber>
      <CellPhone />
   </Employee>
   <Employee>
      <Name>Sally Jones</Name>
      <PhoneNumber>415-555-2000</PhoneNumber>
      <PhoneNumber>415-555-2001</PhoneNumber>
   </Employee>
</Employees>
```

图21-13演示了这个简单XML树的层次结构。

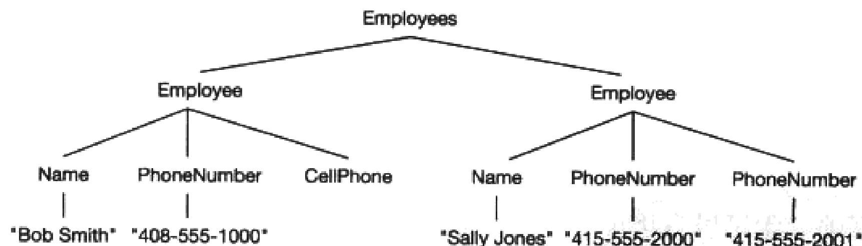



图21-13　示例XML树的层次结构

## 21.7.3　XML 类

LINQ to XML可以以两种方式和XML配合使用。第一种方式是作为简化的XML操作API，第二种方式是使用本章前面看到的LINQ查询工具。我会先从介绍LINQ to XML API开始。

LINQ to XML API由很多表示XML树组件的类组成。我们会使用的最重要的类包括XElement、XAttribute和XDocument。当然，还有其他类，但这些是主要的。

从图21-13中我们可以看到，XML树是一组被嵌套元素。图 21-14 演示了用于构造XML树的类以及它们如何被嵌套。

例如，图21-14显示了如下内容。

- 可作为XDocument节点的直接子节点：
  - 大多数情况下，下面的每一个节点类型各有一个：XDeclaration节点、XDocument节点以及XElement节点。
  - 任何数量的XProcessingInstruction节点。
- 如果在XDocument中有最高级别的XElement节点，那么它就是XML树中其他元素的根。
- 根元素就可以包含任意数量的嵌套XElement、XComment或XProcessingInstruction节点，在任何级别上嵌套。

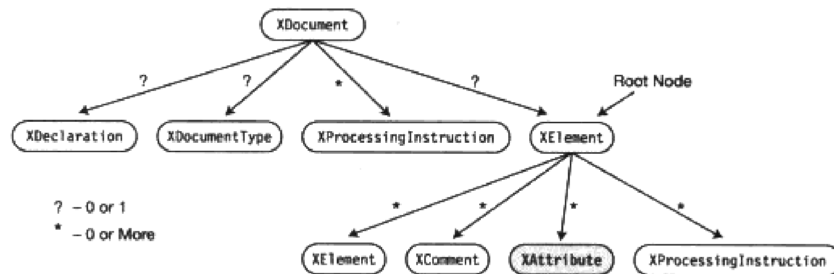



图21-14 XML节点的容器结构

除了XAttribute类，大多数用于创建XML树的类都从一个叫做XNode的类继承。图21-14把XNode类显示为白色背景，XAttribute类显示为灰色背景。

**1. 创建、保存、加载和显示XML文档**

演示XML API的简单性和用途的最好方式就是展示一段简单的示例代码。例如，如下代码演示了使用XML后执行一些重要任务是多么简单。

从创建一个简单的包含名称为Employees节点的XML树开始，两个子节点包含两个职员的名字。注意代码的如下方面：

- 树使用一条语句来创建，在树中就地依次创建所有的嵌套元素，这叫做函数构造（functional construction）。
- 每一个元素被对象创建表达式在适当的位置创建，使用了节点类型的构造函数。

创建树之后，代码使用XDocument的Save方法把它保存在一个叫做EmployeesFile.xml的文件中。然后，再使用XDocument的Load静态方法把XML树从文件中重新读回，并把树赋值给一个新的XDocument对象。最后，使用WriteLine把新的XDocument对象保存的树结构显示出来。

```
using System;
using System.Xml.Linq;                        // 需要的命名空间

class Program {
   static void Main( ) {
      XDocument employees1 =
         new XDocument(                     // 创建XML文档
            new XElement("Employees",       // 创建根元素
```

```
                new XElement("Name", "Bob Smith"),      // 创建元素
                new XElement("Name", "Sally Jones")     // 创建元素
            )
        );

        employees1.Save("EmployeesFile.xml");           // 保存到文件

        // 将保存的文档加载到新变量中
        XDocument employees2 = XDocument.Load("EmployeesFile.xml");
                                            ↑
                                      Static method
        Console.WriteLine(employees2);                  // 显式文档
    }
}
```

这段代码产生了如下的输出：

```
<Employees>
  <Name>Bob Smith</Name>
  <Name>Sally Jones</Name>
</Employees>
```

### 2. 创建XML树

在之前的示例中，我们已经知道了能通过使用XDocument和XElement的构造函数在内存中创建一个XML文档。在这里，两个构造函数：

- 第一个参数都是对象名。
- 第二个参数以及之后的参数包含了XML树的节点。构造函数的第二个参数是一个params参数，也就是说可以有任意多的参数。

例如，如下代码产生了一个XML树并且使用Console.WriteLine方法来显示：

```
using System;
using System.Xml.Linq;                          // 此命名空间是必需的

class Program
{
    static void Main( ) {
        XDocument employeeDoc =
            new XDocument(
                new XElement("Employees",

                    new XElement("Employee",     // 第一个employee元素
                        new XElement("Name", "Bob Smith"),
                        new XElement("PhoneNumber", "408-555-1000") ),

                    new XElement("Employee",     //第二个employee元素
                        new XElement("Name", "Sally Jones"),
                        new XElement("PhoneNumber", "415-555-2000"),
```

```
                    new XElement("PhoneNumber", "415-555-2001") )
            )
        );
        Console.WriteLine(employeeDoc);        // 显示文档
    }
}
```

这段代码产生了如下的输出：

```
<Employees>
  <Employee>
    <Name>Bob Smith</Name>
    <PhoneNumber>408-555-1000</PhoneNumber>
  </Employee>
  <Employee>
    <Name>Sally Jones</Name>
    <PhoneNumber>415-555-2000</PhoneNumber>
    <PhoneNumber>415-555-2001</PhoneNumber>
  </Employee>
</Employees>
```

### 21.7.4 使用 XML 树的值

当我们遍历XML树来获取或修改值时才体现了XML的强大。表21-2给出了用于获取数据的主要方法。

**表21-2 查询XML的方法**

| 方法名称 | 类 | 返回类型 | 描述 |
|---|---|---|---|
| Nodes | XDocument<br>XElement | IEnumerable<object> | 返回当前节点的所有子节点（不管是什么类型） |
| Elements | XDocument<br>XElement | IEnumerable<XElement> | 返回当前节点的XElement子节点，或所有具有某个名字的子节点 |
| Element | XDocument<br>XElement | XElement | 返回当前节点的第一个XElement子节点，或具有某个名字的子节点 |
| Descendants | XElement | IEnumerable<XElement> | 返回所有的XElement子代节点，或所有具有某个名字的XElement子代节点，不管它们处于当前节点下嵌套的什么层次 |
| DescendantsAndSelf | XElement | IEnumerable<XElement> | 和Descendants一样，但是包括当前节点 |
| Ancestors | XElement | IEnumerable<XElement> | 返回所有上级XElement节点，或者所有具有某个名字的上级XElement节点 |
| AncestorsAndSelf | XElement | IEnumerable<XElement> | 和Ancestors一样，但是包括当前节点 |
| Parent | XElement | XElement | 返回当前节点的父节点 |

关于表21-2中的方法，需要了解的一些重要事项如下所示。

- Nodes：Nodes方法返回IEnumerable<object>类型的对象，因为返回的节点可能是不同的类型，比如XElement、XComment等。我们可以使用以类型作为参数的方法OfType(type)来指

定返回某个类型的节点。例如，如下代码只获取XComment节点：

```
IEnumerable<XComment> comments = xd.Nodes().OfType<XComment>();
```

- Elements：由于获取XElements是一个非常普遍的需求，就出现了Nodes.OfType(XElement)()表达式的简短形式——Elements方法。
  - 使用无参数的Elements方法返回所有子XElements。
  - 使用单个name参数的Elements方法只返回具有这个名字的子XElements。例如，如下代码行返回所有具有名字PhoneNumber的子XElement节点。

```
IEnumerable<XElement> empPhones = emp.Elements("PhoneNumber");
```

- Element：这个方法只获取当前节点的第一个子XElement。与Elements方法相似，它可以带一个参数或不带参数调用。如果没有参数，获取第一个子XElement节点。如果带一个参数，它获取第一个具有那个名字的子XElement。
- Descendants和Ancestors：这些方法和Elements以及Parent方法差不多，只不过它们不返回直接的子元素或父元素，而是忽略嵌套级别，包括所有之下或者之上的节点。

如下代码演示了Element和Elements方法：

```
using System;
using System.Collections.Generic;
using System.Xml.Linq;

class Program {
   static void Main( ) {
      XDocument employeeDoc =
         new XDocument(
            new XElement("Employees",
               new XElement("Employee",
                  new XElement("Name", "Bob Smith"),
                  new XElement("PhoneNumber", "408-555-1000")),
               new XElement("Employee",
                  new XElement("Name", "Sally Jones"),
                  new XElement("PhoneNumber", "415-555-2000"),
                  new XElement("PhoneNumber", "415-555-2001"))
               )
            );               获取第一个名为"Employees"的XElement
                                      ↓
      XElement root = employeeDoc.Element("Employees");
      IEnumerable<XElement> employees = root.Elements();

      foreach (XElement emp in employees)
      {                      获取第一个名为"Name"的XElement
                                      ↓
         XElement empNameNode = emp.Element("Name");
         Console.WriteLine(empNameNode.Value);
```

获取所有名为""PhoneNumber"的子元素

```
            IEnumerable<XElement> empPhones = emp.Elements("PhoneNumber");
            foreach (XElement phone in empPhones)
               Console.WriteLine("   {0}", phone.Value);
         }
      }
   }
```

这段代码产生了如下的输出：

```
Bob Smith
   408-555-1000
Sally Jones
   415-555-2000
   415-555-2001
```

## 增加节点以及操作XML

我们可以使用Add方法为现有元素增加子元素。Add方法允许我们在一次方法调用中，不管增加的节点类型是什么，增加希望的任意多一个元素。

例如，如下的代码创建了一个简单的XML树并显示它，然后使用Add方法为根元素增加单个节点。之后，它再次使用Add方法来增加三个元素——两个XElements和一个XComment。注意输出的结果：

```
using System;
using System.Xml.Linq;

class Program
{
   static void Main()
   {
      XDocument xd = new XDocument(               // 创建XML树
         new XElement("root",
            new XElement("first")
         )
      );

      Console.WriteLine("Original tree");
      Console.WriteLine(xd); Console.WriteLine(); // 显示树

      XElement rt = xd.Element("root");           // 获取第一个元素

      rt.Add( new XElement("second"));            // 添加子元素

      rt.Add( new XElement("third"),              // 添加3个或更多子元素
              new XComment("Important Comment"),
```

```
                new XElement("fourth"));

        Console.WriteLine("Modified tree");
        Console.WriteLine(xd);                       // 显示modified tree
    }
}
```

这段代码产生了如下的输出：

```
<root>
  <first />
</root>

<root>
  <first />
  <second />
  <third />
  <!--Important Comment-->
  <fourth />
</root>
```

Add方法把新的子节点放在既有子节点之后，如果我们希望把节点放在子节点之前或者之间也是可以的，使用AddFirst、AddBeforeSelf和AddAfterSelf方法即可。

表21-3列出了最重要的一些操作XML的方法。注意，某些方法针对父节点而其他一些方法针对节点本身。

**表21-3　操作XML的方法**

| 方法名称 | 从哪里调用 | 描　述 |
|---|---|---|
| Add | Parent | 在当前节点的既有子节点后增加新的子节点 |
| AddFirst | Parent | 在当前节点的既有子节点前增加新的子节点 |
| AddBeforeSelf | Node | 在同级别的当前节点之前增加新的节点 |
| AddAfterSelf | Node | 在同级别的当前节点之后增加新的节点 |
| Remove | Node | 删除当前所选的节点及其内容 |
| RemoveNodes | Node | 删除当前所选的XElement及其内容 |
| SetElement | Parent | 设置节点的内容 |
| ReplaceContent | Node | 替换节点的内容 |

## 21.7.5　使用XML属性

属性提供了有关XElement节点的额外信息，它放在XML元素的开始标签中。

当我们以函数方法构造XML树时，可以只需要在XElement的构造函数中包含XAttribute构造函数来增加特性。XAttribute构造函数有两种形式，一种是接受name和value，另一种是接受现有XAttribute的引用。

如下代码为root增加了两个特性。注意，提供给XAttribute构造函数的两个参数都是字符串，

第一个指定了特性名，而第二个指定了值。

```
XDocument xd = new XDocument(
                            名称      值
   new XElement("root",      ↓         ↓
         new XAttribute("color", "red"),
         new XAttribute("size", "large"),
      new XElement("first"),
      new XElement("second")
   )
);

Console.WriteLine(xd);
```

这段代码产生了如下的输出。注意，特性被放在了元素的开始标签中。

```
<root color="red" size="large">
  <first />
  <second />
</root>
```

要从一个XElement节点获取特性可以使用Attribute方法，提供特性名作为参数即可。下面的代码创建了一个在一个节点中有两个特性（color和size）的XML树，然后从特性获取值并且显示它们。

```
static void Main( )
{
   XDocument xd = new XDocument(
      new XElement("root",
         new XAttribute("color", "red"),
         new XAttribute("size", "large"),
         new XElement("first")
      )
   );

   Console.WriteLine(xd); Console.WriteLine();

   XElement rt = xd.Element("root");

   XAttribute color = rt.Attribute("color");
   XAttribute size =  rt.Attribute("size");

   Console.WriteLine("color is {0}", color.Value);
   Console.WriteLine("size  is {0}", size.Value);
}
```

这段代码产生了如下的输出：

```
<root color="red" size="large">
  <first />
</root>

color is red
size  is large
```

要移除特性，我们可以选择一个特性然后使用Remove方法，或在它的父节点中使用SetAttributeValue方法把特性值设置为null。下面是两种方法的演示：

```
static void Main( ) {
   XDocument xd = new XDocument(
      new XElement("root",
         new XAttribute("color", "red"),
         new XAttribute("size", "large"),
         new XElement("first")
      )
   );

   XElement rt = xd.Element("root");

   rt.Attribute("color").Remove();            // 移除color特性
   rt.SetAttributeValue("size", null);        // 移除size特性

   Console.WriteLine(xd);
}
```

这段代码产生了如下的输出：

```
<root>
  <first />
</root>
```

要向XML树增加一个特性或改变特性的值，我们可以使用SetAttributeValue方法，如下代码所示：

```
static void Main( ) {
   XDocument xd = new XDocument(
      new XElement("root",
         new XAttribute("color", "red"),
         new XAttribute("size", "large"),
         new XElement("first")));

   XElement rt = xd.Element("root");

   rt.SetAttributeValue("size",  "medium");      // 改变特性值
   rt.SetAttributeValue("width", "narrow");      // 添加特性

   Console.WriteLine(xd); Console.WriteLine();
}
```

这段代码产生了如下的输出：

```
<root color="red" size="medium" width="narrow">
  <first />
</root>
```

### 21.7.6 节点的其他类型

前面示例中使用的其他三个类型的节点是XComment、XDeclaration以及XProcessingInstruction，见下面部分的描述。

1. XComment

XML注释由<!--和-->记号之间的文本组成。记号之间的文本会被XML解释器忽略。我们可以使用XComment类向一个XML文档插入文本，如下面的代码行所示：

```
new XComment("This is a comment")
```

2. XDeclaration

XML文档从包含XML使用的版本号、使用的字符编码类型以及文档是否依赖外部引用的一行开始。这叫做XML声明，可以使用XDeclaration类来插入，如下代码给出了一个XDeclaration语句的示例：

```
new XDeclaration("1.0", "utf-8", "yes")
```

3. XProcessingInstruction

XML处理指令用于提供XML文档如何被使用和翻译的额外数据，最常见的就是把处理指令用于关联XML文档和一个样式表。

我们可以使用XProcessingInstruction构造函数来包含处理指令。它接受两个字符串参数——目标和数据源。如果处理指令接受多个数据参数，这些参数必须包含在XProcessingInstruction构造函数的第二个字符串参数中，如下面的构造方法代码所示。注意，在这里的示例中，第二个参数是一个字符串，在字符串中的双引号文本使用两个连续的双引号来表现。

```
new XProcessingInstruction( "xml-stylesheet",
                            @"href=""stories"", type=""text/css""")
```

如下代码使用了所有的三个构造函数。

```
static void Main( )
{
   XDocument xd = new XDocument(
      new XDeclaration("1.0", "utf-8", "yes"),
      new XComment("This is a comment"),
      new XProcessingInstruction("xml-stylesheet",
                                 @"href=""stories.css"" type=""text/css"""),
```

```
        new XElement("root",
            new XElement("first"),
            new XElement("second")
        )
    );
}
```

这段代码产生了如下的输出文件。然而，如果使用xd的WriteLine，即使声明语句包含在文档文件中也不会显示。

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--This is a comment-->
<?xml-stylesheet href="stories.css" type="text/css"?>
<root>
  <first />
  <second />
</root>
```

### 21.7.7　使用 LINQ to XML 的 LINQ 查询

现在，我们可以把LINQ XML API和LINQ查询表达式组合在一起来产生简单而强大的XML树搜索。

下面的代码创建了一个简单的XML树，并把它显示在了屏幕上，然后把它保存在一个叫做SimpleSample.xml的文件中。尽管代码没有什么新内容，但是我们会将这个XML树用于之后的示例。

```
using System;
using System.Xml.Linq;

static void Main( )
{
   XDocument xd = new XDocument(
      new XElement("MyElements",
         new XElement("first",
            new XAttribute("color", "red"),
            new XAttribute("size",  "small")),
         new XElement("second",
            new XAttribute("color", "red"),
            new XAttribute("size",  "medium")),
         new XElement("third",
            new XAttribute("color", "blue"),
            new XAttribute("size",  "large"))));

   Console.WriteLine(xd);
   xd.Save("SimpleSample.xml");
}
```

这段代码产生了如下的输出：

```
<MyElements>
  <first color="red" size="small" />
  <second color="red" size="medium" />
  <third color="blue" size="large" />
</MyElements>
```

如下示例代码使用了简单的LINQ查询来从XML中查询节点的子集，然后以各种方式进行显示。这段代码做了如下的事情：

- 它从XML树中选择那些名字有5个字符的元素。由于这些元素的名字是first、second和third，只有first和third这两个名字符合搜索标准，因此这些节点被选中。
- 它显示了所选元素的名字。
- 它格式化并显示了所选节点，包括节点名以及特性值。注意，特性使用Attribute方法来获取，并且特性的值使用Value属性来获取。

```
using System;
using System.Linq;
using System.Xml.Linq;

static void Main( )
{
   XDocument xd = XDocument.Load("SimpleSample.xml"); // 加载文档
   XElement rt = xd.Element("MyElements");           // 获取根元素

   var xyz = from e in rt.Elements()                 // 选择名称包含5个字符的元素
             where e.Name.ToString().Length == 5
             select e;

   foreach (XElement x in xyz)                       // 显式所选的元素
      Console.WriteLine(x.Name.ToString());

   Console.WriteLine();
   foreach (XElement x in xyz)
      Console.WriteLine("Name: {0}, color: {1}, size: {2}",
                        x.Name,
                        x.Attribute("color").Value,
                        x.Attribute("size") .Value);
                            ↑                  ↑
                         获取特性          获取特性的值
}
```

这段代码产生了如下的输出：

```
first
third

Name: first, color: red, size: small
Name: third, color: blue, size: large
```

如下代码使用了一个简单的查询来获取XML树的所有顶层元素，并且为每一个元素创建了一个匿名类型的对象。第一次使用WriteLine方法来显示匿名类型的默认格式，第二次使用WriteLine语句来显示格式化匿名类型对象的成员。

```
using System;
using System.Linq;
using System.Xml.Linq;

static void Main( )
{
   XDocument xd = XDocument.Load("SimpleSample.xml");
   XElement rt = xd.Element("MyElements");

   var xyz = from e in rt.Elements()
             select new { e.Name, color = e.Attribute("color") };
                                     ↑
   foreach (var x in xyz)        创建匿名类型
      Console.WriteLine(x);                           // 默认格式化

   Console.WriteLine();
   foreach (var x in xyz)
      Console.WriteLine("{0,-6},   color: {1, -7}", x.Name, x.color.Value);
}
```

这段代码产生了如下的输出。开始的3行演示了匿名类型的默认格式化，后面3行演示了在第二个WriteLine方法中的格式化字符串中指定的显式格式化。

```
{ Name = first, color = color="red" }
{ Name = second, color = color="red" }
{ Name = third, color = color="blue" }

first ,   color: red
second,   color: red
third ,   color: blue
```

从这些示例中我们可以看到，可以轻易地组合XML API和LINQ查询工具来产生强大的XML查询能力。

# 第 22 章 异步编程简介

**本章内容**

- 进程、线程以及异步编程
- 异步编程模式
- BeginInvoke和EndInvoke
- 计时器

## 22.1 进程、线程以及异步编程

当我们启动一个程序时，系统在内存中创建一个新的进程（process）。进程就是一组资源，它们构成了一个正在运行的程序。这些资源包括虚拟地址空间、文件句柄以及程序启动需要的其他东西的载体。

在进程中，系统创建了一个叫做线程（thread）的内核对象，线程体现了一个程序的真实执行情况。（线程是执行线程的缩写。）一旦程序准备完毕，系统在线程中开始执行Main方法的第一条语句。

有关线程的一些重要事项如下：

- 默认情况下，一个进程只包含一个线程，它从程序开始执行一直到程序结束。
- 一个线程可以发起另外一个线程，因此，在任何时候，一个进程都可能在不同状态有多个线程来执行程序的不同部分。
- 如果一个进程中有多个线程，它们会分享进程的资源。
- 线程是由系统负责调度的处理器（而不是进程）上的执行单元。

到目前为止，我们在书中使用的所有示例程序都只有一个线程，并且从程序的第一行语句到最后一行语句顺序执行，这叫做同步编程。异步编程指的是程序发起多个线程，它们在理论上是在同一时间执行的。（其实不一定真的在同一时间执行。）

如果程序运行在一个多处理器的系统上，不同的线程实际上可能在不同的处理器上同时执行。这可以大幅度提高性能，由于多核处理器的普及，我们需要写程序来利用这个机会。

然而，在单处理器的系统上，同一时间内处理器肯定只能执行一个指令。这种情况下，操作系统就会协调线程，使处理器在它们之间分享。每一个线程都可以获取称作时间片（time slice）的一小段时间，这种轮流分享处理器的方式使所有线程能通过代码按照它们自己的方式进行工作。

### 22.1.1 多线程处理带来的问题

在程序中使用多个线程叫做多线程处理，它增加了程序的负荷和额外的复杂度，例如：

- 创建和销毁线程都有时间和资源的成本；
- 安排线程、把它们加载到处理器中以及在每次时间片段之后保存状态都需要时间；
- 由于进程中的线程共享相同的资源和堆，这就需要增加额外的编程复杂度来确保它们之间的工作不相干扰；
- 调试多线程程序非常困难，因为程序每次运行的时间不同可能就会产生不同的结果，所以在调试器中运行程序就是在浪费时间。

除这些问题外，多线程的好处相对这些成本来说是值得的——只要它被合理利用而不是被过度使用。例如，我们已经知道，在多处理器的系统上，如果不同的线程运行在不同的处理器上，执行效率会高很多。

为了帮我们降低创建和销毁线程相关的成本，CLR为每一个进程维护了一个线程池（thread pool）。一开始进程的线程池是空的，如果进程使用的线程被创建，并且完成了线程的执行，它不会被销毁，而是加入到进程的线程池中。之后，如果进程需要另外一个线程，CLR就会从池中还原一个线程，这就节省了很多时间。

另外一个常见的示例是，多线程处理对GUI编程来说很重要。用户希望在他点击了按钮或使用键盘之后能有一个快速的响应，如果程序需要执行耗费一段时间的操作，就可以把它放在另外一个线程中进行，保持主线程用于对用户的输入进行反馈。如果程序在那段时间中没有响应，这是完全不可接受的。

### 22.1.2　多线程处理的复杂度

尽管从概念上说多线程处理不是很复杂，但是要让一个复杂程序在所有细节上都很完美是比较困难的，需要在如下方面进行考虑。

- 线程之间的通信：有很多内建的机制能用于线程之间通信，因此通常只要使用内存就能完成了，对于同一个进程的所有线程来说，内存是可见的也是可访问的。
- 协调线程：尽管创建线程很简单，我们还需要能协调它们的行为。例如，一个线程可能需要在继续执行之前等待另外一个线程结束。
- 同步资源使用：由于进程内的所有线程共享相同的资源和内存，我们需要确保不同的线程没有在同一时间访问和改变它们引起冲突的状态。

System.Threading命名空间包含的一些类和类型使我们可以创建复杂的多线程程序。包括Thread类本身以及诸如Mutex、Semaphore和Monitor等用来同步资源使用的类。它们的复杂使用以及区别等细节已经超出了本书的范围，建议读者找相关的书籍进行深入研究。

然而，我们可以通过两种简单的技术为程序增加非常强大的多线程处理——异步委托和计时器，在本章剩余部分会介绍这些内容。对于大多数程序来说，它们可能是我们需要的最好的技术。

## 22.2　异步编程模式

到现在为止，在本书中见到的代码都是同步的。然而，C#可以有一个简单易用的机制用于异步执行方法，那就是使用委托。

在第15章中，我们介绍了委托的主题，并且了解到当委托对象调用时，它调用了它的调用列表中包含的方法。就像程序调用方法一样，这是同步完成的。

如果委托对象在调用列表中只有一个方法（之后会叫做引用方法），它就可以异步执行这个方法。委托类有两个方法，叫做BeginInvoke和EndInvoke，它们就是用来这么做的。这些方法以如下方式使用：

- 当我们调用委托的BeginInvoke方法时，它开始在线程池中的独立线程上执行引用方法，并且立即返回原始线程。原始线程可以继续，而引用方法会在线程池的线程中并行执行。
- 当程序希望获取已完成的异步方法的结果时，可以检查BeginInvoke返回的IAsyncResult的IsCompleted属性，或调用委托的EndInvoke方法来等待委托完成。

图22-1演示了使用这三种标准模式的过程。对于这三种模式来说，原始线程都发起了一个异步方法，然后做一些其他处理。然而，这些模式不同的是，原始线程获取发起的线程已经完成的消息的方式。

- 在等待一直到完成（wait-until-done）模式中，在发起了异步方法以及做了一些其他处理之后，原始线程就中断并且等异步方法完成之后再继续。
- 在轮询（polling）模式中，原始线程定期检查发起的线程是否完成，如果没有则可以继续做一些其他的事情。
- 在回调（callback）模式中，原始线程一直执行，无需等待或检查发起的线程是否完成。在发起的线程中的引用方法完成之后，发起的线程就会调用回调方法，由回调方法在调用EndInvoke之前处理异步方法的结构。

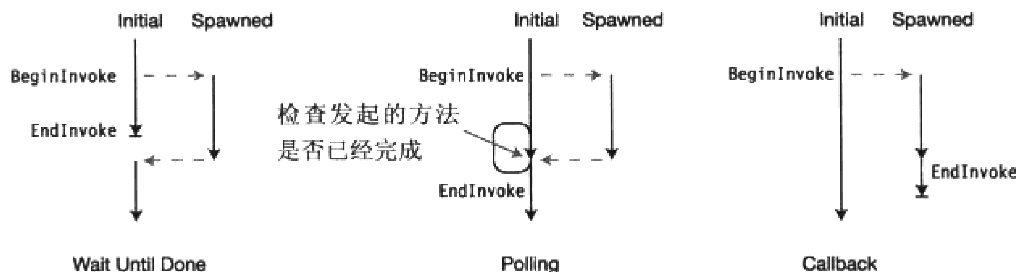



图22-1 异步方法调用的标准模式

## 22.3 BeginInvoke 和 EndInvoke

在学习异步编程模式的示例之前，让我们先研究一下BeginInvoke和EndInvoke方法。一些需要了解的有关BeginInvoke的重要事项如下。

- 在调用BeginInvoke时，参数列表中的实参组成如下：
  - 引用方法需要的参数；
  - 两个额外的参数——callback参数和state参数。
- BeginInvoke从线程池中获取一个线程并且在新的线程开始时运行引用方法。
- BeginInvoke返回给调用线程一个实现IAsyncResult接口的对象。这个接口引用包含了异

步方法的当前状态，原始线程然后可以继续执行。

如下的代码给出了一个调用委托的BeginInvoke方法的示例。第一行声明了叫做MyDel的委托类型。下一行声明了一个和委托匹配的叫做Sum的方法。

- 之后的行声明了一个叫做del的Mydel委托类型的委托对象，并且使用Sum方法来初始化它的调用列表。
- 最后一行代码调用了委托对象的BeginInvoke方法并且提供了两个委托参数3和5，以及两个BeginInvoke的参数callback和state，在本例中都设为null。执行后，BeginInfoke方法进行两个操作：
  - 从线程池中获取一个线程并且在新的线程上开始运行Sum方法，提供它3和5作为参数。
  - 它收集新线程的状态信息并且把IAsyncResult接口的引用返回给调用线程来提供这些数据。调用线程把它保存在一个叫做iar的变量中。

```
delegate long MyDel( int first, int second ); //委托声明
   ...
static long Sum(int x, int y){ ... }          //方法匹配委托
   ...
MyDel del = new MyDel(Sum);                   // 创建委托对象
IAsyncResult iar = del.BeginInvoke( 3, 5, null, null );
        ↑                  ↑           ↑        ↑
  有关新线程的信息      异步调用委托    委托参数   额外参数
```

EndInvoke方法用来获取由异步方法调用返回的值，并且释放线程使用的资源。EndInvoke有如下的特性。

- 它接受一个由BeginInvoke方法返回的IAsyncResult对象的引用，并找到它关联的线程。
- 如果线程池的线程已经退出，EndInvoke做如下的事情：
  - 它清理退出线程的状态并且释放它的资源。
  - 它找到引用方法返回的值并且把它的值作为返回值。
- 如果当EndInvoke被调用时线程池的线程仍然在运行，调用线程就会停止并等待，直到清理完毕并返回值。因为EndInvoke是为开启的线程进行清理，所以必须确保对每一个BeginInvoke都调用EndInvoke。
- 如果异步方法触发了异常，在调用EndInvoke时会抛出异常。

如下的代码行给出了一个调用EndInvoke并从异步方法获取值的示例。我们必须把IAsyncResult对象的引用作为参数。

```
                  委托对象
                     ↓
long result = del.EndInvoke( iar );
      ↑                       ↑
异步方法的返回值         IAsyncResult对象
```

EndInvoke提供了从异步方法调用的所有输出，包括ref和out参数。如果委托的引用方法有ref或out参数，它们必须包含在EndInvoke的参数列表中，并且在IAsyncResult对象引用之前，如下所示：

```
long result = del.EndInvoke(out someInt, iar);
   ↑                            ↑         ↑
异步方法的返回值                 out参数   IAsyncResult对象
```

## 22.3.1 等待-直到结束模式

现在，我们已经理解了BeginInvoke和EndInvoke方法，那么就让我们来看看异步编程模式吧。我们要学习的第一种异步编程模式是等待—直到结束模式。在这种模式里，原始线程发起一个异步方法的调用，做一些其他处理，然后停止并等待，直到开启的线程结束。它总结如下：

```
IAsyncResult iar = del.BeginInvoke( 3, 5, null, null );
     // Do additional work in the calling thread, while the method
     // is being executed asynchronously in the spawned thread.
     ...
long result = del.EndInvoke( iar );
```

如下代码给出了一个使用这种模式的完整示例。代码使用Thread类的Sleep方法将它自己挂起0.1秒。Thread类在System.Threading命名空间下。

```
using System.Threading;
   ...
delegate long MyDel( int first, int second );    // 声明委托类型

class Program
{
   static long Sum(int x, int y)                  // 声明异步方法
   {
      Console.WriteLine("                         Inside Sum");
      Thread.Sleep(100);

      return x + y;
   }

   static void Main( )
   {
      MyDel del = new MyDel(Sum);

      Console.WriteLine( "Before BeginInvoke" );
      IAsyncResult iar = del.BeginInvoke(3, 5, null, null); // 开始异步调用
      Console.WriteLine( "After  BeginInvoke" );

      Console.WriteLine( "Doing stuff" );

      long result = del.EndInvoke( iar );    // 等待结束并获取结果
```

```
        Console.WriteLine( "After  EndInvoke: {0}", result );
    }
}
```

这段代码产生了如下的输出：

```
Before BeginInvoke
After  BeginInvoke
Doing stuff
                        Inside Sum
After  EndInvoke: 8
```

## 22.3.2 AsyncResult 类

既然我们已经在看到了BeginInvoke和EndInvoke的最简单形式，是时候来进一步接触IASyncResult了。它是使用这些方法的必要部分。

BeginInvoke返回一个IASyncResult接口的引用（内部是AsyncResult类的对象）。AsyncResult类表现了异步方法的状态。图22-2演示了这个类中的一些重要部分。有关这个类的重要事项如下：

- 当我们调用委托对象的BeginInvoke方法时，系统创建了一个AsyncResult类的对象。然而，它不返回类对象的引用，而是返回对象中包含的IAsyncResult接口的引用。
- AsyncResult对象包含一个叫做AsyncDelegate的属性，它返回一个指向被调用来开启异步方法的委托的引用。但是，这个属性是类对象的一部分而不是接口的一部分。
- IsCompleted属性返回一个布尔值，表示异步方法是否完成。
- AsyncState属性返回一个对象的引用，它被作为BeginInvoke方法调用时的state参数。它返回object类型的引用，我们会在22.3.4节中解释这部分内容。

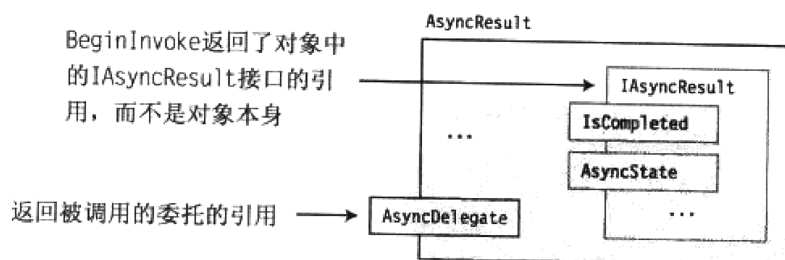



图22-2　AsyncResult类对象

## 22.3.3 轮询模式

在轮询模式中，原始线程发起了异步方法的调用，做一些其他事情，然后使用IAsyncResult对象的IsComplete属性来定期检查开启的线程是否完成。如果异步方法已经完成，原始线程就调用EndInvoke并继续。否则，它做一些其他处理，然后再检查。这里的“处理”仅仅是由0数到10,000,000。

```
delegate long MyDel(int first, int second);

class Program
{
   static long Sum(int x, int y)
   {
      Console.WriteLine("                       Inside Sum");
      Thread.Sleep(100);

      return x + y;
   }

   static void Main()
   {
      MyDel del = new MyDel(Sum);  发起异步方法
                                        ↓
      IAsyncResult iar = del.BeginInvoke(3, 5, null, null);
      Console.WriteLine("After BeginInvoke");
               检查异步方法是否完成
                      ↓
      while ( !iar.IsCompleted )
      {
         Console.WriteLine("Not Done");

         // 继续处理
         for (long i = 0; i < 10000000; i++)
            ;
      }
      Console.WriteLine("Done");
               调用EndInvoke来获取接口并进行清理
                           ↓
      long result = del.EndInvoke(iar);
      Console.WriteLine("Result: {0}", result);
   }
}
```

这段代码产生了如下的输出：

```
After BeginInvoke
Not Done
                   Inside Sum
Not Done
Not Done
Done
Result: 8
```

## 22.3.4 回调模式

在之前的等待一直到结束模式以及轮询模式中，初始线程继续它自己的控制流程，直到它知

道开启的线程完成。然后，它获取结果并继续。

回调模式的不同之处在于，一旦初始线程发起了异步方法，它就自己管自己了，不再考虑同步。当异步方法调用结束之后，系统调用一个用户自定义的方法来处理结果，并且调用委托的EndInvoke方法。这个用户自定义的方法叫做回调方法或回调。

BeginInvoke的参数列表中最后的两个额外参数被回调方法用作：

- 第一个参数，callback参数，是回调方法的名字。
- 第二个参数，state参数，可以是null或要传入回调方法的一个对象的引用。我们可以通过使用IAsyncResult参数的AsyncState属性来获取这个对象，参数的类型是object。

### 1. 回调方法

回调方法的签名和返回类型必须和AsyncCallback委托类型所描述的形式一致。它需要方法接受一个IAsyncResult作为参数并且返回类型是void，如下所示：

```
void AsyncCallback( IAsyncResult iar )
```

我们有多种方式可以为BeginInvoke方法提供回调方法。由于BeginInvoke中的callback参数是AsyncCallback类型的委托，我们可以以委托形式提供，如下面的第一行代码所示。或者，我们也可以只提供回调方法名称，让编译器为我们创建委托，两种形式是完全等价的。

```
                                使用回调方法创建委托
IAsyncResult iar1 =                    ↓
   del.BeginInvoke(3, 5, new AsyncCallback(CallWhenDone), null);
                                    只需要用回调方法的名字
                                            ↓
IAsyncResult iar2 = del.BeginInvoke(3, 5, CallWhenDone, null);
```

第二个BeginInvoke的参数是发送给回调方法的对象。它可以是任何类型的对象，但是参数类型是object，所以在回调方法中，我们必须转换成正确的类型。

### 2. 在回调方法内调用EndInvoke

在回调方法内，我们的代码应该调用委托的EndInvoke方法来处理异步方法执行后的输出值。要调用委托的EndInvoke方法，我们肯定需要委托对象的引用，而它在初始线程中，不在开启的线程中。

如果我们不使用BeginInvoke的state参数作其他用途，可以使用它发送委托的引用给回调方法，如下所示：

```
                   委托对象                  把委托对象作为状态参数
                      ↓                                ↓
IAsyncResult iar = del.BeginInvoke(3, 5, CallWhenDone, del);
```

否则，我们可以从发送给方法作为参数的IAsyncResult对象中提取出委托的引用。如图22-3以及下面的代码所示：

- 给回调方法的参数只有一个，就是刚结束的异步方法的IAsyncResult接口的引用。请记住，

IAsyncResult接口对象在AsyncResult类对象的内部。

- 尽管IAsyncResult接口没有委托对象的引用，而包含它的AsyncResult类对象却有委托对象的引用。所以，示例代码方法体的第一行就通过转换接口引用为类类型来获取类对象的引用。变量ar现在就有类对象的引用。
- 有了类对象的引用，我们现在就可以调用类对象的AsyncDelegate属性并且把它转化为合适的委托类型。这样就得到了委托引用，我们可以用它来调用EndInvoke。

```
using System.Runtime.Remoting.Messaging;     // 包含AsyncResult类

void CallWhenDone( IAsyncResult iar )
{
   AsyncResult ar = (AsyncResult) iar;         // 获取类对象的引用
   MyDel del = (MyDel) ar.AsyncDelegate;       // 获取委托的引用

   long Sum = del.EndInvoke( iar );            // 调用EndInvoke
      ...
}
```

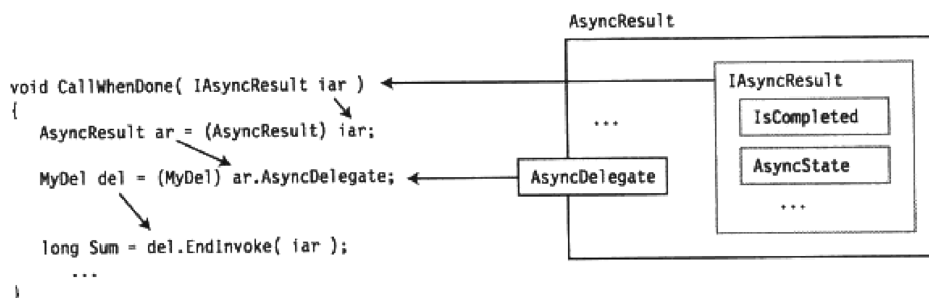



图22-3 从回调方法内部提取出委托的引用

如下代码把所有知识点放在了一起，给出了一个使用回调模式的示例。

```
using System.Runtime.Remoting.Messaging;  // 调用AsyncResult类型
   ...
delegate long MyDel(int first, int second);

class Program {
   static long Sum(int x, int y)
   {
      Console.WriteLine("                        Inside Sum");
      Thread.Sleep(100);
      return x + y;
   }

   static void CallWhenDone(IAsyncResult iar) {
      Console.WriteLine("                        Inside CallWhenDone.");
      AsyncResult ar = (AsyncResult) iar;
      MyDel del = (MyDel)ar.AsyncDelegate;
```

```
        long result = del.EndInvoke(iar);
        Console.WriteLine
          ("                     The result is: {0}.", result);
    }

    static void Main() {
        MyDel del = new MyDel(Sum);

        Console.WriteLine("Before BeginInvoke");
        IAsyncResult iar =
          del.BeginInvoke(3, 5, new AsyncCallback(CallWhenDone), null);

        Console.WriteLine("Doing more work in Main.");
        Thread.Sleep(500);
        Console.WriteLine("Done with Main. Exiting.");
    }
}
```

这段代码产生了如下的输出：

```
Before BeginInvoke
Doing more work in Main.
                     Inside Sum
                     Inside CallWhenDone.
                     The result is: 8.
Done with Main. Exiting.
```

## 22.4 计时器

计时器提供了另外一种正规的重复运行异步方法的方法。尽管在.NET BCL中有好几个可用的Timer类，但在这里我们只会介绍System.Threading命名空间中的一个。

有关计时器类需要了解的重要事项如下所示。

- 计时器在每次时间到期之后调用回调方法。回调方法必须是TimerCallback委托形式的，结构如下所示。它接受了一个object类型作为参数，并且返回类型是void。

```
void TimerCallback( object state )
```

- 当计时器到期之后，系统会从线程池中的线程上开启一个回调方法，提供state对象作为其参数，并且开始运行。
- 我们可以设置的计时器的一些特性如下：
  - dueTime是回调方法首次被调用之前的时间。如果dueTime被设置为特殊的值Timeout.Infinite，则计时器不会开始。如果被设置为0，回调函数会被立即调用。
  - period是两次成功调用回调函数之间的时间间隔。如果它的值设置为Timeout.Infinite，回调在首次被调用之后不会再被调用。

- state可以是null或在每次回调方法执行时要传入的对象的引用。

Timer类的构造函数接受回调方法名称、dueTime、period以及state作为参数。Timer有很多构造函数，最为常用的形式如下：

```
Timer( TimerCallback callback, object state, uint dueTime, uint period)
```

如下代码语句展示了一个创建Timer对象的示例：

```
                    回调的名字        在2 000毫秒后第一次调用
                        ↓                         ↓
Timer myTimer = new Timer ( MyCallback, someObject, 2000, 1000 );
                                          ↑              ↑
                                   传给回调的对象    每1 000毫秒
                                                     调用一次
```

一旦Timer对象被创建，我们可以使用Change方法来改变它的dueTime或period方法。

如下代码给出了一个使用计时器的示例。Main方法创建了一个计时器，2秒钟之后它会首次调用回调，然后每隔1秒再调用1次。回调方法只是输出了包含它被调用的次数的消息。

```
using System;
using System.Threading;

namespace Timers
{
   class Program
   {
      int TimesCalled = 0;

      void Display (object state)
      {
         Console.WriteLine("{0} {1}",(string)state, ++TimesCalled);
      }

      static void Main( )
      {
         Program p = new Program();
                                                  2秒后第一次调用
         Timer myTimer = new Timer                       ↓
               (p.Display, "Processing timer event", 2000, 1000);
         Console.WriteLine("Timer started.");                ↑
                                                      每一秒重复一次
         Console.ReadLine();
      }
   }
}
```

这段代码在被关闭前的5秒内产生了如下的输出：

```
Timer started.
Processing timer event 1
Processing timer event 2
Processing timer event 3
Processing timer event 4
```

.NET BCL还提供了几个其他计时器类，每一个都有其用途。其他计时器类如下所示。

- System.Windows.Forms.Timer：这个类在Windows应用程序中使用，用来定期把WM_TIMER消息放到程序的消息队列中。当程序从队列获取消息后，它会在主用户接口线程中异步处理，这对Windows应用程序来说非常重要。
- System.Timers.Timer：这个类更复杂，它包含了很多成员，使我们可以通过属性和方法来操作计时器。它还有一个叫做Elapsed的成员事件，每次时间到期就会发起这个事件。这个计时器可以运行在用户接口线程或工作者线程上。

第 23 章

# 预处理指令

**本章内容**

- 什么是预处理指令
- 基本规则
- #define和#undef指令
- 条件编译
- 条件编译结构
- 诊断指令
- 行号指令
- 区域指令
- #pragma warning指令

## 23.1 什么是预处理指令

源代码指定了程序的定义。**预处理指令**（preprocessor directive）指示编译器如何处理源代码。例如，在某些情况下，我们可能希望编译器忽略一部分代码；而在其他情况下，我们可能希望代码被编译。预处理指令给了我们这样的选项。

在C和C++中有实际的预处理阶段，此时预处理程序遍历源代码并且为之后的编译阶段准备文本输出流，在C#中没有实际的预处理程序。“预处理”指令由编译器来处理，而这个术语保留了下来。

## 23.2 基本规则

下面是预处理指令的最重要的一些语法规则：

- 预处理指令必须和C#代码在不同的行。
- 与C#语句不同，预处理指令不需要以分号结尾。
- 包含预处理指令的每一行必须以#字符开始。
  - 在#字符前可以有空格。
  - 在#字符和指令之间可以有空格。

- 允许行尾注释。
- 在预处理指令所在的行不允许分隔符注释。

这里的一些示例阐释了这些规则：

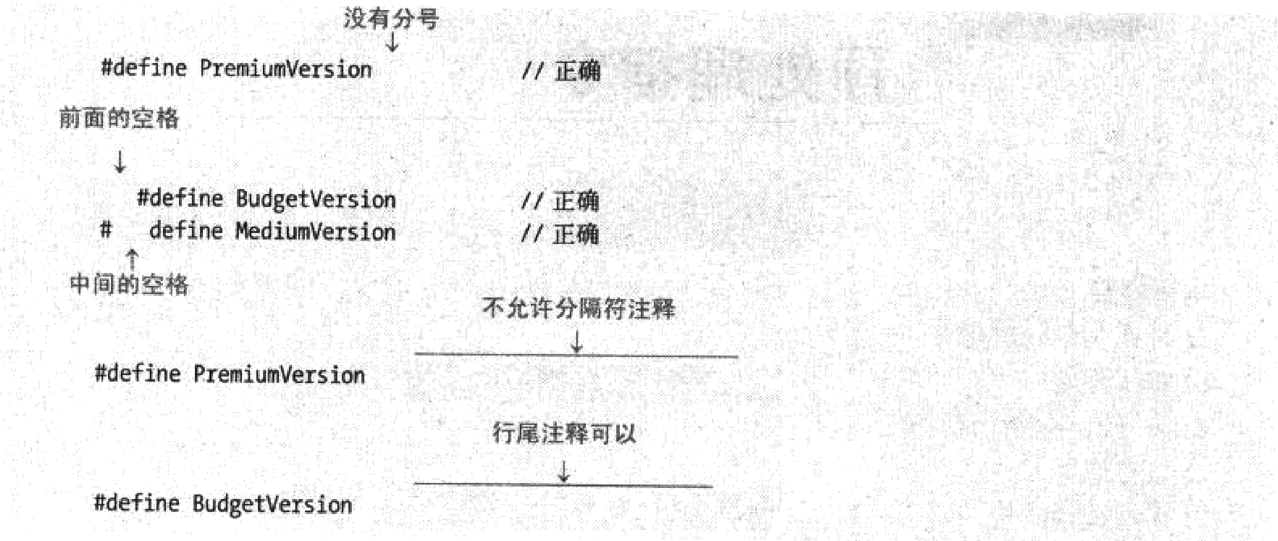



表23-1列出了预处理指令。

表23-1　预处理指令

| 指　令 | 含义概要 |
|---|---|
| #define identifier | 定义编译符 |
| #undef identifier | 取消定义编译符 |
| #if expression | 如果表达式是true，编译下面的片段 |
| #elif expression | 如果表达式是true，编译下面的片段 |
| #else | 如果之前的#if或#elif表达式是false，编译下面的片段 |
| #endif | 标记为一个#if结构的结束 |
| #region name | 标记一段代码的开始，没有编译效果 |
| #endregion name | 标记一段代码的结束，没有编译效果 |
| #warning message | 显示编译时的警告消息 |
| #error message | 显示编译时的错误消息 |
| #line indicator | 修改在编译器消息中显示的行数 |
| #pragma text | 指定有关程序上下文的信息 |

## 23.3　#define 和#undef 指令

编译符号是只有两种状态的标识符，要么被定义，要么未被定义。编译符号有如下的特性：

- 它可以是除了true或false以外的任何标识符，包括C#关键词，以及在C#代码中声明的标

识符，这两者都是可以的。

- 它没有值。与C和C++不同，它不表示字符串。

如表23-1所示：

- #define指令声明一个编译符号。
- #undef指令取消定义一个编译符号。

```
#define PremiumVersion
#define EconomyVersion
   ...
#undef PremiumVersion
```

#define和#undef指令只能在源文件的第一行，也就是任何C#代码之前使用。在C#代码开始后，#define和#undef指令就不能再使用。

```
using System;                          // C#代码的第1行
#define PremiumVersion                 // 错误

namespace Eagle
{
   #define PremiumVersion              // 错误
   ...
```

编译符号的范围被限制于单个源文件。只要编译符号在任何C#代码之前，重复定义已存在的编译符号也是允许的。

```
#define AValue
#define BValue

#define AValue                         // 重复定义
```

## 23.4 条件编译

条件编译允许我们根据某个编译符号是否被定义标注一段代码被编译或跳过。

有四个指令可以用来指定条件编译：

- #if
- #else
- #elif
- #endif

条件是一个返回true或false的简单表达式。

- 如表23-2所总结的，条件可以由单个编译符号、符号表达式或操作符组成。子条件可以使用圆括号分组。
- 文本true或false也可以在条件表达式中使用。

表23-2　在#if和#elif指令中使用的条件

| 参数类型 | 意　义 | 运算结果 |
| --- | --- | --- |
| 编译符号 | 使用#define指令（未）定义的标识符 | True：如果符号已经使用#define指令定义<br>False：其他 |
| 表达式 | 使用符号和操作符!、==、!=、$$、\|\|构建的 | True：如果表达式运算结果为true<br>False：其他 |

如下是一个条件编译的示例：

```
          Expression
#if !DemoVersion
   ...
#endif                  Expression
#if (LeftHanded && OemVersion) || FullVersion
   ...
#endif

#if true    // 下面的代码片段总是会被编译
   ...
#endif
```

## 23.5　条件编译结构

#if和#endif指令在条件编译结构中需要配对使用。只要有#if指令，就必须有配对的#endif。#if和#if…#else结构如图23-1所示。

- 如果#if结构中的条件运算结果为true，随后的代码段就会被编译，否则就会被跳过。
- 在#if…#else结构中，如果条件运算结果为true，CodeSection1就会被编译；否则，CodeSection2会被编译。




图23-1　#if和#else结构

例如，如下的代码演示了简单的#if…#else结构。如果符号RightHanded被定义了，那么#if和#else之间的代码会被编译。否则，#else和#endif之间的代码会被编译。

```
...
#if RightHanded
   // 实现右边函数的代码
```

```
    ...
#else
    // 实现左边函数的代码
    ...
#endif
```

图23-2演示了#if…#else以及#if…#elif…#else的结构。

- 在#if…#else结构中，如果Cond1运算结果为true，CodeSection1就会被编译，然后就会继续编译#endif之后的代码。
    - 否则，如果Cond2运算结果为true，CodeSection2就会被编译，然后会继续编译#endif之后的代码。
    - 直到条件运算结果为true或所有条件都返回false。如果这样，结构中没有任何代码段会被编译，会继续编译#endif之后的代码。
- #if…#elif…#else结构也是相同的工作方式，只不过没有条件是true的情况下，会编译#else之后的代码段，然后会继续编译#endif之后的代码。




图23-2 #elif结构

如下的代码演示了#if…#elif…#else结构。包含程序版本描述的字符串根据定义的编译符号被设置为各种值。

```
#define DemoVersionWithoutTimeLimit
    ...
    const int intExpireLength = 30;
    string strVersionDesc = null;
    int    intExpireCount = 0;

#if   DemoVersionWithTimeLimit
    intExpireCount = intExpireLength;
    strVersionDesc = "This version of Supergame Plus will expire in 30 days";

#elif DemoVersionWithoutTimeLimit
    strVersionDesc = "Demo Version of Supergame Plus";

#elif OEMVersion
    strVersionDesc = " Supergame Plus, distributed under license";
```

```
#else
   strVersionDesc = "The original Supergame Plus!!";

#endif

   Console.WriteLine( strVersionDesc );
      ...
```

## 23.6　诊断指令

诊断指令产生用户自定义的编译时警告或错误消息。

下面是诊断指令的语法。Message是字符串，但是需要注意，与普通的C#字符串不同，它们不需要被引号包围。

```
#warning Message

#error Message
```

当编译器遇到诊断指令时，它会输出相关的消息。诊断指令的消息会和任何编译器产生的警告和错误消息列在一起。

例如，如下代码显示了一个#error指令和一个#warning指令。

- #error指令在#if结构中，因此只有符合#if指令的条件时才会生成消息。
- #warning指令用于提醒程序员回头来清理这段代码。

```
#define RightHanded
#define LeftHanded

#if RightHanded && LeftHanded
#error Can't build for both RightHanded and LeftHanded
#endif

#warning Remember to come back and clean up this code!
```

## 23.7　行号指令

行号指令可以做很多事情，诸如：

- 改变由编译器警告和错误消息报告的出现行数。
- 改变被编译源文件的文件名。
- 对交互式调试器隐藏一些行。

#line指令的语法如下：

```
#line integer            // 设置下一行值为整数的行的行号
#line "filename"         // 设置文件名
#line default            // 重新保存实际的行号和文件名
```

```
#line hidden            // 在断点调试器中隐藏代码
#line                   // 停止在调试器中隐藏代码
```

#line指令加上一个整数参数会使编译器认为下面代码的行是所设置的行，之后的行数会根据这行的数字继续。

- 要改变外观文件名，可以在双引号内使用文件名作为参数。双引号是必须的。
- 要返回真实行号和真实文件名，可以使用default参数。
- 要对交互调试器的断点调试功能隐藏代码段，可以使用hidden作为参数。要停止隐藏，可以使用不带任何参数的指令。到目前为止，这个功能大多用于在ASP.NET中隐藏编译器生成的代码。

下面的代码给出了行号指令的示例：

```
#line 226
      x = y + z; // 编译器将执行第226行
      ...

#line 330 "SourceFile.cs" // 改变报告的行号和文件名
      var1 = var2 + var3;
      ...

#line default               // 重新保存行号和文件名
```

## 23.8 区域指令

区域指令允许我们标注和有选择性地命名一段代码。#region指令：

- 被放置在希望标注的代码段之上；
- 用指令后的可选字符串文本作为其名字；
- 在之后的代码中必须由#endregion指令终止。

尽管区域指令被编译器忽略，但它们可以被源代码工具所使用。例如，Visual Studio允许我们很简单地隐藏或显示区域。

作为示例，下面的代码中有一个叫做Constructors的区域，它封闭了MyClass类的两个构造函数。在Visual Studio中，如果不想看到其中的代码，我们可以把这个区域折叠成一行，如果又想对它进行操作或增加另外一个构造函数，还可以扩展它。

```
#region Constructors
   MyClass()
   { ... }

   MyClass(string s)
   { ... }
#endregion
```

如图23-3所示，区域可以被嵌套。

```
static void Main( )
{
      ...
#region first
 
#region second
   ...
#endregion
 
#region third
   ...
#endregion
 
#endregion
}
```

图23-3　嵌套的区域

## 23.9　#pragma warning 指令

#pragma warning指令允许我们关闭或重新开启警告消息。

- 要关闭警告消息，可以使用disable加上逗号分隔的希望关闭的警告数列表的形式。
- 要重新开启警告消息，可以使用restore加上逗号分隔的希望关闭的警告数列表的形式。

例如，下面的代码关闭了两个警告消息：618和414。在后面的代码中又开启了618警告消息，但还是保持414消息为关闭状态。

```
                    要关闭的警告消息
                           ↓
#pragma warning disable 618, 414
   ··· 列出的警告消息在这段代码中处于关闭状态
#pragma warning restore 618
```

如果我们使用任一种不带警告数字列表的形式，这个命令会应用于所有警告。例如，下面的代码关闭，然后恢复所有警告消息。

```
#pragma warning disable
      ··· 所有警告消息在这段代码中处于关闭状态

#pragma warning restore
      ··· 所有警告消息在这段代码中处于开启状态
```

# 第 24 章 反射和特性

**本章内容**

- 元数据和反射
- Type类
- 获取Type对象
- 什么是特性
- 应用特性
- 预定义的保留的特性
- 有关应用特性的更多内容
- 自定义特性
- 访问特性

## 24.1 元数据和反射

大多数程序都是用来处理数据的。它们读、写、操作和显示数据。（图形也是一种数据的形式。）程序员为某种目的创建和使用一些类型，因此，在设计时必须理解所使用的类型的特性。

然而，对于某些程序来说，它们操作的数据不是数字、文本或图形，而是程序和程序类型本身的信息。

- 有关程序及其类型的数据被称作元数据（metadata），它们保存在程序的程序集中。
- 程序在运行时，可以查看其他程序集或其本身的元数据。一个运行的程序查看本身的元数据或其他程序的元数据的行为叫做反射（reflection）。

对象浏览器是显示元数据的程序的一个示例。它可以读取程序集，然后显示所包含的类型以及类型的所有特性和成员。

本章将介绍程序如何使用Type类来反射数据，以及程序员如何使用特性来给类型添加元数据。

**说明** 要使用反射，我们必须使用System.Reflection命名空间。

## 24.2 Type 类

之前已经介绍了如何声明和使用C#中的类型。包括预定义类型（int、long和string等）、BCL

中的类型（Console、IEnumerable等）以及用户自定义类型（MyClass、Mydel等）。每一种类型都有自己的成员和特性。

BCL声明了一个叫做Type的抽象类，它被设计用来包含类型的特性。使用这个类的对象能让我们获取程序使用的类型的信息。

由于Type是抽象类，因此它不能有实例。而是在运行时，CLR创建从Type（RuntimeType）派生的类的实例，Type包含了类型信息。当我们要访问这些实例时，CLR不会返回派生类的引用而是Type基类的引用。但是，为了简单起见，在本章剩余的篇幅中，我会把引用所指向的对象称为Type类型的对象（虽然从技术角度来说是一个引用了Type类型的派生类型的对象）。

需要了解的有关Type的重要事项如下：

- 对于程序中用到的每一个类型，CLR都会创建一个包含这个类型信息的Type类型的对象。
- 程序中用到的每一个类型都会关联到独立的Type类的对象。
- 不管创建的类型有多少个实例，只有一个Type对象会关联到这些实例。

图24-1显示了一个运行的程序，它有两个MyClass对象和一个OtherClass对象。注意，尽管有两个MyClass的实例，只会有一个Type对象来表示它。

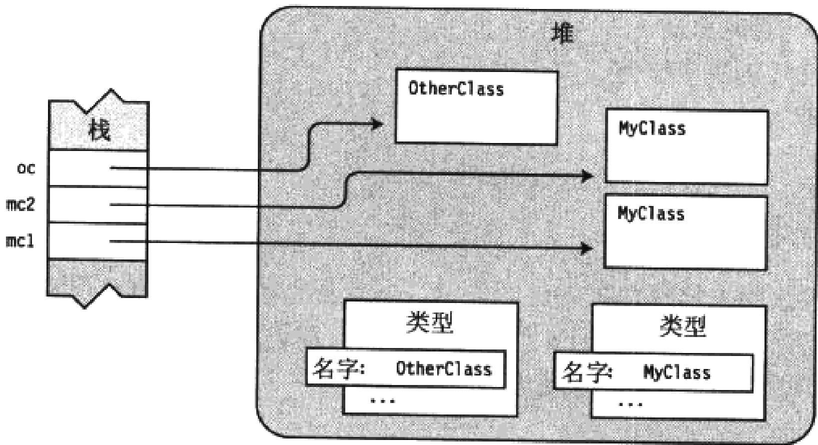



图24-1　对于程序中使用的每一个类型CLR都会实例化Type类型的对象

我们可以从Type对象中获取需要了解的有关类型的几乎所有信息。表 24-1列出了类中更有用的成员。

表24-1　System.Type类精选的成员

| 成　员 | 成员类型 | 描　述 |
|---|---|---|
| Name | 属性 | 返回类型的名字 |
| Namespace | 属性 | 返回包含类型声明的命名空间 |
| GetFields | 方法 | 返回类型的字段列表 |
| GetProperties | 方法 | 返回类型的属性列表 |
| GetMethods | 方法 | 返回类型的方法列表 |

## 24.3　获取 Type 对象

有好几种方式可以获取Type对象。我们将学习使用GetType方法和typeof运算符来获取Type对象。

object类型包含了一个叫做GetType的方法，它返回对实例的Type对象的引用。由于每一个类型最终都是从object继承的，所以我们可以在任何类型对象上使用GetType方法来获取它的Type对象，如下所示：

```
Type t = myInstance.GetType();
```

下面的代码演示了如何声明一个基类以及从它派生的子类。Main方法创建了每一个类的实例并且把这些引用放在了一个叫做bca的数组中以方便使用。在外层的foreach循环中，代码得到了Type对象并且输出类的名字，然后获取类的字段并输出。图24-2演示了内存中的对象。

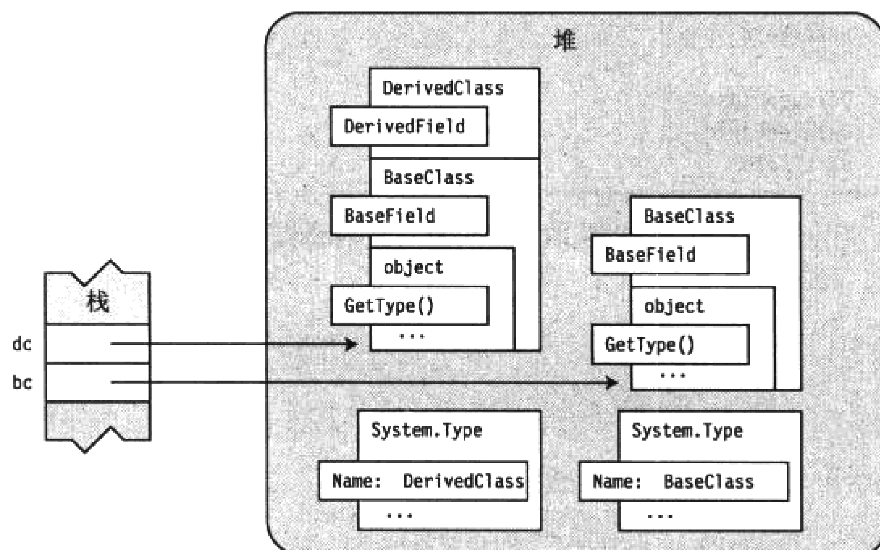



图24-2　基类和派生类对象以及它们的Type对象

```
class BaseClass
{ public int BaseField = 0; }

class DerivedClass : BaseClass
{ public int DerivedField = 0; }

class Program
{
   static void Main( )
   {
      var bc = new BaseClass();
      var dc = new DerivedClass();
      BaseClass[] bca = new BaseClass[] { bc, dc };
```

```
            foreach (var v in bca)
            {
               Type t = v.GetType();

               Console.WriteLine("Object type : {0}", t.Name);

               FieldInfo[] fi = t.GetFields();
               foreach (var f in fi)
                  Console.WriteLine("      Field : {0}", f.Name);
               Console.WriteLine();
            }
         }
      }
```

这段代码产生了如下的输出：

```
Object type : BaseClass
      Field : BaseField

Object type : DerivedClass
      Field : DerivedField
      Field : BaseField
```

我们还可以使用typeof运算符来获取Type对象。只需要提供类型名作为操作数，它就会返回Type对象的引用，如下所示：

```
Type t = typeof( DerivedClass );
           ↑          ↑
        运算符     希望的Type对象
```

下面的代码给出了一个使用typeof运算符的简单示例：

```
using System;
using System.Reflection;                         // 必须使用该命名空间

namespace SimpleReflection
{
   class BaseClass
   { public int MyFieldBase; }

   class DerivedClass : BaseClass
   { public int MyFieldDerived; }

   class Program
   {
      static void Main( )
      {
         Type tbc = typeof(DerivedClass);
         Console.WriteLine("Result is {0}.", tbc.Name);
```

```
            Console.WriteLine("It has the following fields:");
            FieldInfo[] fi = tbc.GetFields();
            foreach (var f in fi)
               Console.WriteLine("   {0}", f.Name);
         }
      }
   }
```

这段代码产生了如下的输出：

```
Result is DerivedClass.
It has the following fields:
   MyFieldDerived
   MyFieldBase
```

## 24.4 什么是特性

特性（attribute）是一种允许我们向程序的程序集增加元数据的语言结构。它是用于保存程序结构信息的某种特殊类型的类。

- 将应用了特性的程序结构（program construct）叫做目标（target）。
- 设计用来获取和使用元数据的程序（比如对象浏览器）被叫做特性的消费者（consumer）。
- .NET预定了很多特性，我们也可以声明自定义特性。

图24-3是使用特性中相关组件的概览，并且也演示了如下有关特性的要点：

- 我们在源代码中将特性应用于程序结构。
- 编译器获取源代码并且从特性产生元数据，然后把元数据放到程序集中。
- 消费者程序可以获取特性的元数据以及程序中其他组件的元数据。注意，编译器同时生产和消费特性。

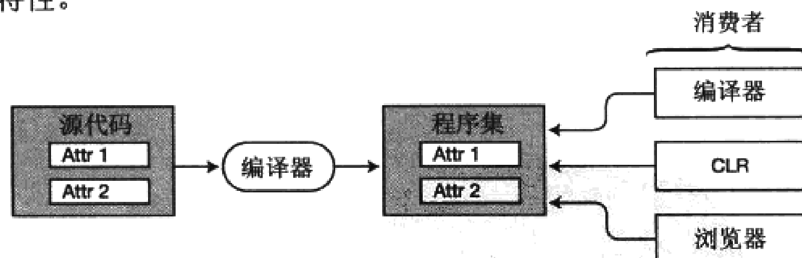



图24-3 创建和使用特性的相关组件

根据惯例，特性名使用Pascal命名法并且以Attribute后缀结尾。当为目标应用特性时，我们可以不使用后缀。例如，对于SerializableAttribute和MyAttributeAttribute这两个特性，我们在把它们应用到结构时可以使用Serializable和MyAttribute短名称。

## 24.5 应用特性

特性的目的是告诉编译器把程序结构的某组元数据嵌入程序集。我们可以通过把特性应用到

结构来实现。

- 在结构前放置特性片段来应用特性。
- 特性片段被方括号包围，其中是特性名和特性的参数列表。

例如，下面的代码演示了两个类的开始部分。最初的几行代码演示了把一个叫做Serializable的特性应用到MyClass。注意，Serializable没有参数列表。第二个类的声明有一个叫做MyAttribute的特性，它有一个带有两个string参数的参数列表。

```
[ Serializable ]                                 // 特性
public class MyClass
{ ...

[ MyAttribute("Simple class", "Version 3.57") ]  // 带有参数的特性
public class MyOtherClass
{ ...
```

一些有关特性的需要了解的重要事项如下：

- 大多数特性只针对直接跟随在一个或多个特性片段后的结构。
- 应用了特性的结构称作被特性装饰（decorated或adorned，两者没有什么区别）。

## 24.6 预定义的保留的特性

在学习如何定义自己的特性之前，本小节会先介绍.NET的两个预定义的、保留的特性：Obsolete和Conditional特性。

### 24.6.1 Obsolete 特性

Obsolete特性允许我们将程序结构标注为过期的并且在代码编译时显示有用的警告消息。以下代码给出了一个使用的示例：

```
class Program            应用特性
{                           ↓
    [Obsolete("Use method SuperPrintOut")]   // 将特性应用到方法
    static void PrintOut(string str)
    {
        Console.WriteLine(str);
    }

    static void Main(string[] args)
    {
        PrintOut("Start of Main");           // 调用obsolete方法
    }
}
```

注意，即使PrintOut被标注为过期，Main方法还是调用了它。代码编译也运行得很好并且产生了如下的输出：

```
Start of Main
```

不过，在编译的过程中，编译器产生了下面的CS0618警告消息来通知我们正在使用一个过期的结构：

```
'AttrObs.Program.PrintOut(string)' is obsolete: 'Use method SuperPrintOut'
```

另外一个Obsolete特性的重载接受了bool类型的第二个参数。这个参数指定目标是否应该被标记为错误而不仅仅是警告。以下代码指定了它需要被标记为错误：

```
                                标记为错误
                                    ↓
[ Obsolete("Use method SuperPrintOut", true) ]
static void PrintOut(string str)
{ ...
```

## 24.6.2 Conditional 特性

Conditional特性允许我们包括或排斥某个特定方法的所有调用。为方法声明应用Conditional特性并把编译符作为参数来使用。

- ❑ 如果定义了编译符号，那么编译器会包含所有调用这个方法的代码，这和普通方法没有什么区别。
- ❑ 如果没有定义编译符号，那么编译器会忽略代码中这个方法的所有调用。

定义方法的CIL代码本身会包含在程序集中。只是调用代码会被插入或忽略。

例如，在如下的代码中，把Conditional特性应用到对一个叫做TraceMessage的方法的声明上。特性只有一个参数，在这里是字符串DoTrace。

- ❑ 当编译器编译这段代码时，它会检查是否有一个编译符号被定义为DoTrace。
- ❑ 如果DoTrace被定义，编译器就会像往常一样包含所有对TraceMessage方法的调用。
- ❑ 如果没有DoTrace这样的编译符号被定义，编译器就不会输出任何对TraceMessage的调用代码。

```
                编译符号
                   ↓
[Conditional( "DoTrace" )]
static void TraceMessage(string str)
{
   Console.WriteLine(str);
}
```

Conditional特性的示例

以下代码演示了一个使用Conditional特性的完整示例。

- ❑ Main方法包含了两个对TraceMessage方法的调用。

- TraceMessage方法的声明被用Conditional特性装饰，它带有DoTrace编译符号作为参数。因此，如果DoTrace被定义，那么编译器就会包含所有对TraceMessage的调用代码。
- 由于代码的第一行定义了叫做DoTrace的编译符，编译器会包含两个对TraceMessage的调用。

```
#define DoTrace
using System;
using System.Diagnostics;

namespace AttributesConditional
{
   class Program
   {
      [Conditional( "DoTrace" )]
      static void TraceMessage(string str)
      { Console.WriteLine(str); }

      static void Main( )
      {
         TraceMessage("Start of Main");
         Console.WriteLine("Doing work in Main.");
         TraceMessage("End of Main");
      }
   }
}
```

这段代码产生了如下的输出：

```
Start of Main
Doing work in Main.
End of Main
```

如果注释了第一行来取消DoTrace的定义，编译器就不再会插入两次对TraceMessage的调用代码。这次，如果我们运行程序，就会产生如下的输出：

```
Doing work in Main.
```

## 24.6.3　预定义的特性

.NET框架预定了很多编译器和CLR能理解和解释的特性，表24-2 列出了一些。在表中使用了不带Attribute后缀的短名称。例如，CLSCompliant的全名是ClSCompliantAttribute。

**表24-2　定义在.NET中的重要特性**

| 特　性 | 意　义 |
|---|---|
| CLSCompliant | 声明公共暴露的成员应该被编译器检测是否符合CLS。兼容的程序集可以被任何.NET-兼容语言使用 |

（续）

| 特 性 | 意 义 |
|---|---|
| Serializable | 声明结构可以被序列化 |
| NonSerialized | 声明结构不能被序列化 |
| Obsolete | 声明结构不应该再使用。如果使用结构，编译器还会产生编译时警告或者错误消息 |
| DLLImport | 声明是非托管代码实现的 |
| WebMethod | 声明方法应该被作为XML Web服务的一部分暴露 |
| AttributeUsage | 声明特性能应用到什么类型的程序结构。将这个特性应用到特性声明上 |

## 24.7 有关应用特性的更多内容

至此，我们演示了特性的简单使用，都是为方法应用单个特性。这部分内容将会讲述其他特性的使用方式。

### 24.7.1 多个特性

我们可以为单个结构应用多个特性。

- 多个特性可以使用下面列出的任何一种格式：
  - 独立的特性片段相互叠在一起；
  - 单个特性片段，特性之间使用逗号分隔。
- 我们可以以任何次序列出特性。

例如，下面的两个代码片段显示了应用多个特性的两种方式。两个片段的代码是等价的。

```
[ Serializable ]                                          // Stacked
[ MyAttribute("Simple class", "Version 3.57") ]

[ MyAttribute("Simple class", "Version 3.57"), Serializable ]    // Commas
                 ↑                                  ↑
                特性                               特性
```

### 24.7.2 其他类型的目标

除了类，我们还可以将特性应用到诸如字段和属性等其他程序结构。以下的声明显示了字段上的特性以及方法上的多个特性：

```
[MyAttribute("Holds a value", "Version 3.2")]            // 字段上的特性
public int MyField;

[Obsolete]                                               // 方法上的特性
[MyAttribute("Prints out a message.", "Version 3.6")]
public void PrintOut()
{
   ...
```

我们还可以显式地标注特性，从而将它应用到特殊的目标结构。要使用显式目标，在特性片段的开始处放置目标类型，后面跟冒号。例如，如下的代码用特性装饰方法，并且还把特性应用到返回值上。

```
显式目标
   ↓
[method: MyAttribute("Prints out a message.", "Version 3.6")]
[return: MyAttribute("This value represents ...", "Version 2.3")]
public long ReturnSetting()
{
   ...
```

如表24-3所列，C#语言定义了10个标准的特性目标。大多数目标名可以自说明(self-explanatory)，而type覆盖了类、结构、委托、枚举和结构。typevar目标名称指定使用泛型结构的类型参数。

表24-3　特性目标

| | | | |
|---|---|---|---|
| event | field | type | typevar |
| method | param | assembly | module |
| property | return | | |

### 24.7.3　全局特性

我们还可以通过使用assembly和module来使用显式目标把特性设置在程序集或模块级别。(程序集和模块在第10章中解释过。)一些有关程序集级别的特性的要点如下：

- 程序级级别的特性必须放置在任何命名空间之外，并且通常放置在AssemblyInfo.cs文件中。
- AssemblyInfo.cs文件通常包含有关公司、产品以及版权信息的元数据。

如下的代码行摘自AssemblyInfo.cs文件：

```
[assembly: AssemblyTitle("SuperWidget")]
[assembly: AssemblyDescription("Implements the SuperWidget product.")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("McArthur Widgets, Inc.")]
[assembly: AssemblyProduct("Super Widget Deluxe")]
[assembly: AssemblyCopyright("Copyright © McArthur Widgets 2008")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

## 24.8　自定义特性

你或许已经注意到了，应用特性的语法和之前见过的其他语法很不相同。你可能会觉得特性是和结构完全不同的类型，其实不是，特性只是某个特殊类型的类。

有关特性类的一些要点如下：

- 用户自定义的特性类叫做自定义特性。

❑ 所有特性类都派生自System.Attribute。

### 24.8.1 声明自定义特性

总体来说，声明一个特性类和声明其他类一样。然而，有一些事项值得注意，如下所示。

❑ 要声明一个自定义特性，需要做如下工作：
  - 声明一个派生自System.Attribute的类。
  - 给它起一个以后缀Attribute结尾的名字。

❑ 为了安全，通常建议你声明一个sealed的特性类。

例如，下面的代码显示了MyAttributeAttribute特性的声明的开始部分：



```
public sealed class MyAttributeAttribute : System.Attribute
{
   ...
```



由于特性持有目标的信息，所有特性类的公共成员只能是：

❑ 字段
❑ 属性
❑ 构造函数

### 24.8.2 使用特性构造函数

特性和其他类一样，都有构造函数。每一个特性至少必须有一个公共构造函数。

❑ 和其他类一样，如果你不声明构造函数，编译器会为我们产生一个隐式的、公共的、无参的构造函数。

❑ 特性的构造函数和其他构造函数一样，可以被重载。

❑ 声明构造函数时必须使用类全名，包括后缀。我们只可以在应用特性时使用短名称。

例如，如果有如下的构造函数（名字没有包含后缀），编译器会产生一个错误消息：



```
public MyAttributeAttribute(string desc, string ver)
{
   Description   = desc;
   VersionNumber = ver;
}
```

### 24.8.3 指定构造函数

当我们为目标应用特性时，其实是在指定应该使用哪个构造函数来创建特性的实例。列在特性应用中的参数其实就是构造函数的参数。

例如，在下面的代码中，MyAttribute被应用到一个字段和一个方法上。对于字段，声明指定了使用单个字符串的构造函数。对于方法，声明指定了使用两个字符串的构造函数。

```
[MyAttribute("Holds a value")]                // 使用一个字符串的构造函数
public int MyField;

[MyAttribute("Version 1.3", "Sal Martin")]    // 使用两个字符串的构造函数
public void MyMethod()
{ ...
```

其他有关特性构造函数的要点如下：

- 在应用特性时，构造函数的实参必须是在编译期能确定值的常量表达式。
- 如果应用的特性构造函数没有参数，可以省略圆括号。例如，如下代码的两个类都使用MyAttr特性的无参构造函数。两种形式的意义是相同的。

```
[MyAttr]
class SomeClass ...

[MyAttr()]
class OtherClass ...
```

### 24.8.4 使用构造函数

和其他类一样，我们不能显式调用构造函数。特性的实例创建后就会调用构造函数，只有特性的消费者才能访问特性。这一点与其他类的实例很不相同，这些实例都创建在使用对象创建表达式的位置。应用一个特性是一条声明语句，它不会决定什么时候构造特性类的对象。

图24-4比较了普通类构造函数的使用和特性的构造函数的使用。

- 命令语句的实际意义是："在这里创建新的类"。
- 声明语句的意义是："这个特性和这个目标相关联，如果需要构造特性，使用这个构造函数"。

```
MyClass mc = new MyClass("Hello", 15);     |     [MyAttribute("Holds a value")]
```

命令语句 | 声明语句

图24-4 比较构造函数的使用

### 24.8.5 构造函数中的位置参数和命名参数

至此，我们见到的特性构造函数中的参数和普通类构造函数的参数差不多。与普通类型的构造函数一样，特性构造函数的实参必须有正确的次序，而且要与类定义中的形参相匹配。编译器能根据参数列表中参数的位置知道哪个实参和哪个形参相匹配，因此这种参数叫做位置参数。

但是，特性的构造函数还有另外一种类型的实参，叫做命名参数。

- 命名参数设置特性的字段或属性的值。
- 命名参数由字段或属性名后接等号和初始化值构成。

命名参数是实参，它的声明和构造函数的形参是一样的。唯一的区别是特性被应用时提供的实参列表。

如下代码显示了使用一个位置参数和两个命名参数来应用一个特性：

```
              位置参数               命名参数            命名参数
                 ↓                      ↓                   ↓
[MyAttribute("An excellent class", Reviewer="Amy McArthur", Ver="0.7.15.33")]
                                         ↑                  ↑
                                        等号               等号
```

下面的代码演示了特性类的声明以及为MyClass类应用特性。注意，构造函数的声明只列出了一个形参，但我们可通过命名参数给构造函数三个实参。两个命名参数设置了字段Ver和Reviewer的值。

```
public sealed class MyAttributeAttribute : System.Attribute
{
    public string Description;
    public string Ver;
    public string Reviewer;

    public MyAttributeAttribute(string desc)  // 一个形参
    { Description = desc; }
}
                                   三个参数
                                      ↓
[MyAttribute("An excellent class", Reviewer="Amy McArthur", Ver="7.15.33")]
class MyClass
{ ... }
```

**说明** 构造函数需要的任何位置参数都必须放在命名参数之前。

## 24.8.6 限制特性的使用

我们已经看到了可以为类应用特性。而特性本身就是类，有一个很重要的预定义特性可以用来应用到自定义特性上，那就是AttributeUsage特性。我们可以使用它来限制特性使用在某个目标类型上。

例如，如果我们希望自定义特性MyAttribute只能应用到方法上，那么可以以如下形式使用AttributeUsage：

```
                      只针对方法
                          ↓
[ AttributeUsage( AttributeTarget.Method ) ]
public sealed class MyAttributeAttribute : System.Attribute
{ ...
```

AttributeUsage有三个重要的公共属性，如表24-4所示。表中显示了属性名和属性的含义。对于后两个属性，还显示了它们的默认值。

表24-4　AttributeUsage的公共属性

| 名　字 | 意　义 | 默 认 值 |
|---|---|---|
| ValidOn | 保存特性能应用到的目标类型的列表。构造函数的第一个参数必须是AttributeTarget类型的枚举值 | |
| Inherited | 一个布尔值，它指示特性是否会被装饰类型的派生类所继承 | true |
| AllowMutiple | 一个指示目标是否被应用多个特性的实例的布尔值 | false |

AttributeUsage**的构造函数**

AttributeUsage的构造函数接受单个位置参数，该参数指定了特性允许的目标类型。它用这个参数来设置ValidOn属性，可接受目标类型是AttributeTarget枚举的成员。AttributeTarget枚举的完整成员列表如表 24-5所示。

我们可以通过使用按位或运算符来组合使用类型。例如，在下面的代码中，被装饰的特性只能应用到方法和构造函数上。

目标

```
[ AttributeUsage( AttributeTarget.Method | AttributeTarget.Constructor ) ]
public sealed class MyAttributeAttribute : System.Attribute
{ ...
```

表24-5　AttributeTarget枚举的成员

| All | Assembly | Class | Constructor |
|---|---|---|---|
| Delegate | Enum | Event | Field |
| GenericParameter | Interface | Method | Module |
| Parameter | Property | ReturnValue | Struct |

当我们为特性声明应用AttributeUsage时，构造函数至少需要一个参数，参数包含的目标类型会保存在ValidOn中。我们还可以通过使用命名参数有选择性地设置Inherited和AllowMultiple属性。如果我们不设置，它们会保持如表24-4所示的默认值。

作为示例，下面一段代码指定了MyAttribute的如下方面：

- MyAttribute能且只能应用到类上。
- MyAttribute不会被应用它的派生类所继承。
- 不能有MyAttribute的多个实例应用到同一个目标上。

```
[ AttributeUsage( AttributeTarget.Class,          // 必需的
                  Inherited = false,              // 可选的
                  AllowMultiple = false ) ]       // 可选的
public sealed class MyAttributeAttribute : System.Attribute
{ ...
```

### 24.8.7　自定义特性的最佳实践

在写自定义特性时，如下的实践是被强烈推荐的：

- 特性类应该表示目标结构的一些状态。
- 如果特性必需某些字段，可以通过包含具有位置参数的构造函数来收集数据，可选字段可以采用命名参数按需初始化。
- 除了属性之外，不要实现公共方法或其他函数成员。
- 为了更安全，把特性类声明为sealed。
- 在特性声明中使用AttributeUsage来显式指定特性目标组。

如下代码演示了这些准则：

```
[AttributeUsage( AttributeTargets.Class )]
public sealed class  MyAttributeAttribute : System.Attribute
{
   private string _Description;
   private string _VersionNumber;
   private string _ReviewerID;

   public string Description
      { get { return _Description; }   set { _Description = value; } }

   public string VersionNumber
      { get { return _VersionNumber; } set { _VersionNumber = value; } }

   public string ReviewerID
      { get { return _ReviewerID; }    set { _ReviewerID = value; } }

   public MyAttributeAttribute(string desc, string ver)
   {
      Description   = desc;
      VersionNumber = ver;
   }
}
```

## 24.9 访问特性

在本章开始处，我们已经看到了可以使用Type对象来获取类型信息。对于访问自定义特性来说，我们也可以这么做。Type的两个方法在这里非常有用：IsDefined和GetCustomAttribute。

### 24.9.1 使用 IsDefined 方法

我们可以使用Type对象的IsDefined方法来检测某个特性是否应用到了某个类上。

例如，以下的代码声明了一个有特性的类MyClass，并且作为自己特性的消费者在程序中访问声明和被应用的特性。在代码的开始处是MyAttribute特性和应用特性的MyClass类的声明。这段代码做了下面的事情：

- 首先，Main创建了类的一个对象。然后通过使用从object基类继承的GetType方法获取了Type对象的一个引用。

- 有了Type对象的引用，就可以调用IsDefined方法来判断MyAttribute特性是否应用到了这个类。
  - 第一个参数接受需要检查的特性的Type对象。
  - 第二个参数是bool类型的，它指示是否搜索MyClass的继承树来查找这个特性。

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class MyAttributeAttribute : System.Attribute
{ ... }

[MyAttribute("Check it out", "2.4")]
class MyClass {  }

class Program {
   static void Main() {
      MyClass mc = new MyClass();
      Type t = mc.GetType();
      bool isDefined =             // 创建特性的类型
         t.IsDefined(typeof(MyAttributeAttribute), false);

      if( isDefined )
         Console.WriteLine("MyAttribute is applied to type {0}", t.Name);
   }
}
```

这段代码产生了如下的输出：

```
MyAttribute is applied to type MyClass
```

### 24.9.2　使用 GetCustomAttribute 方法

GetCustomAttributes方法返回应用到结构的特性的数组。

- 实际返回的对象是object的数组，因此我们必须将它强制转换为相应的特性类型。
- 布尔参数指定是否搜索继承树来查找特性。

```
object[] AttArr = t.GetCustomAttributes(false);
```

- 当GetCustomAttributes方法调用后，每一个与目标相关联的特性的实例就会被创建。

下面的代码使用了前面的示例中相同的特性和类声明。但是，在这种情况下，它不检测特性是否应用到了类，而是获取应用到类的特性的数组，然后遍历它们，输出它们的成员的值。

```
static void Main()
{
   Type t = typeof(MyClass);
   object[] AttArr = t.GetCustomAttributes(false);

   foreach (Attribute a in AttArr)
```

```
    {
        MyAttributeAttribute attr = a as MyAttributeAttribute;
        if (null != attr)
        {
            Console.WriteLine("Description    : {0}", attr.Description);
            Console.WriteLine("Version Number : {0}", attr.VersionNumber);
            Console.WriteLine("Reviewer ID    : {0}", attr.ReviewerID);
        }
    }
}
```

这段代码产生了如下的输出：

```
Description    : Check it out
Version Number : 2.4
Reviewer ID    :
```

# 第 25 章 其他主题

**本章内容**

- 概述
- 字符串
- 把字符串解析为数据值
- 可空类型
- Main方法
- 文档注释
- 嵌套类型

## 25.1 概述

在本章中，我会介绍使用C#时的一些重要而又不适合放到其他章节的其他主题。它们包括字符串操作、可空类型、Main方法、文档注释以及嵌套类型。

## 25.2 字符串

对于内部计算来说0和1很适合，但是对于人类可读的输入和输出，我们需要字符串。BCL提供了很多能让字符串操作变得更简单的类。

C#预定义的string类型代表了.NET的System.String类。对于字符串，最需要理解的概念如下：

- 字符串是Unicode字符串数组。
- 字符串是不可变的（immutable）—— 它们不能被修改。

string类型有很多有用的字符串操作成员，包括允许我们进行诸如检测长度、改变大小写、连接字符串等一些有用的操作。表25-1列出了其中一些最有用的成员。

**表25-1 string类型的有用成员**

| 成　员 | 类　型 | 意　义 |
|---|---|---|
| Length | 属性 | 返回字符串长度 |
| Concat | 静态方法 | 返回连接参数字符串后的字符串 |
| Contains | 方法 | 返回指示参数是否是对象字符串的子字符串的bool值 |

（续）

| 成员 | 类型 | 意义 |
| --- | --- | --- |
| Format | 静态方法 | 返回格式化后的字符串 |
| Remove | 方法 | 从对象字符串中移除一组字符 |
| Replace | 方法 | 替换对象字符串中的字符 |
| SubString | 方法 | 获取对象字符串的子字符串 |
| ToUpper | 方法 | 返回对象字符串的副本，其中所有字母的字符都为大写 |
| ToLower | 方法 | 返回对象字符串的副本，其中所有字母的字符都为小写 |

从表25-1中的大多数方法的名字来看，好像它们都会改变字符串对象。其实，它们不会改变字符串而是返回了新的副本。对于一个string，任何“改变”都会分配一个新的恒定字符串。

例如，下面的代码声明并初始化了一个叫做s的字符串。第一个WriteLine语句调用了s的ToUpper方法，它返回了字符串中所有字母为大写形式的副本。最后一行输出了s的值，可以看到，字符串并没有改变。

```
string s = "Hi there.";

Console.WriteLine("{0}", s.ToUpper());          // 输出所有字母为大写的副本
Console.WriteLine("{0}", s);                    // 字符串没有变
```

这段代码产生了如下的输出：

```
HI THERE.
Hi there.
```

### 25.2.1 使用 StringBuilder 类

StringBuilder类可以产生能被修改的字符串。

- StringBuilder类是BCL的成员，位于System.Text命名空间中。
- StringBuilder对象是Unicode字符的可变数组。

例如，下面的代码声明并初始化了一个StringBuilder类型的字符串，然后输出了它的值。第四行代码通过替换字符串的一部分改变了其实际对象。当输出它的值时，我们可以看到，和string类型的对象不同，StringBuilder对象确实被修改了。

```
using System.Text;

StringBuilder sb = new StringBuilder("Hi there.");
Console.WriteLine("{0}", sb);                  // 输出字符串
sb.Replace("Hi", "Hello");                     // 替换子字符串
Console.WriteLine("{0}", sb);                  // 输出改变后的字符串
```

这段代码产生了如下的输出：

```
Hi there.
Hello there.
```

当创建了StringBuilder对象之后，类分配了一个比当前字符串长度更长的缓冲区。只要缓冲区能容纳对字符串的改变就不会分配新的内存。如果对字符串的改变需要的空间比缓冲中的可用空间多，就会分配更大的缓冲区，并把字符串拷贝到其中。和原来的缓冲区一样，新的缓冲区也有额外的空间。

要获取StringBuilder对应的字符串内容，我们只需要简单调用它的ToString方法即可。

### 25.2.2　格式化数字字符串

贯穿本书的示例代码一直使用WriteLine方法来显示值。每次，我们都使用由大括号包围整数组成的简单替代标记形式。然而在很多时候，我们更希望以更合适的格式而不是一个简单的数字来表现文本字符串的输出。例如，可能会希望把值作为货币或者某个小数位数的定点值来显示。这些都可以通过格式化字符串来实现。

例如，下面的代码由两条打印值500的语句组成。第一行没有使用任何其他格式化来打印数字，而第二行的格式化字符串指定了数字应该被格式化成货币。

```
Console.WriteLine("The value: {0}."  , 500);       // 输出数字
Console.WriteLine("The value: {0:C}.", 500);       // 格式为货币
                                 ↑
                             格式化为货币
```

这段代码产生了如下的输出：

```
The value: 500.
The value: $500.00.
```

两条语句的不同之处在于，格式项以格式说明符形式包括了额外的信息。大括号内的格式化说明符的语法由三个字段组成：索引号、对齐说明符和格式说明符。语法如图25-1所示。

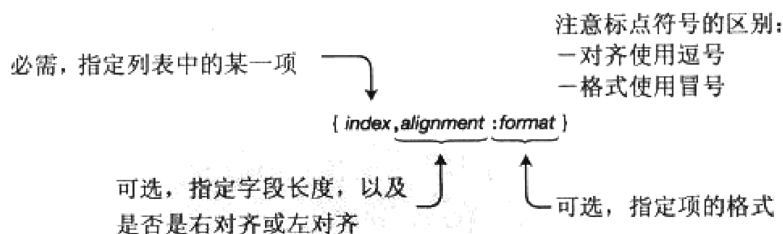



图25-1　格式项的语法

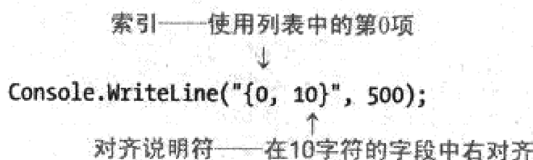
格式项首先需要索引号。现在我们已经知道，索引指定了之后的格式化字符串应该格式化列表中的哪一项。索引号是必需的，并且列表项的数字必须从0开始。

#### 1. 对齐说明符

对齐说明符表示了字段中字符的最小宽度。对齐说明符有如下特性：

- 它是可选的，并且使用逗号来和索引号分离。
- 它由一个正整数或负整数组成。
  - 整数表示了字段使用字符的最少数量。

■ 符号表示了右对齐或左对齐。正数表示右对齐，负数表示左对齐。




例如，如下代码使用了两个格式项来格式化int型变量myInt的值。在第一个示例中，myInt的值以在10个字符的字符串中右对齐的形式进行显示；第二个示例中则是左对齐。格式项放在两个竖杠中间，仅仅用于在输出中显示它们的左右界限。

```
int myInt = 500;
Console.WriteLine("|{0, 10}|", myInt);                 // 右对齐
Console.WriteLine("|{0,-10}|", myInt);                 // 左对齐
```

这段代码产生了如下的输出，在两个竖杠的中间有10个字符：

```
|       500|
|500       |
```

值的实际表示可能会比对齐说明符指定的字符数多一些或少一些：

- 如果要表示的字符数比对齐说明符中指定的字符数少，那么其余字符会使用空格填充。
- 如果要表示的字符数多于指定的字符数，对齐说明符会被忽略，并且使用所有字符进行表示。

**2. 格式组件**

格式组件指定了数字应该以哪种形式表示。例如，应该被当作货币、十进制数字、十六进制数字还是定点符号来表示？

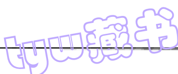
格式组件有两部分，如图25-2所示：

- 格式说明符是一个字母的字符，是9个内置字符格式之一。字符可以是大写或小写形式。大小写对于某些说明符来说比较重要，而对于另外一些说明符来说则不重要。
- 精度说明符是可选的，由1～2位数字组成。它的实际意义取决于格式说明符。




图25-2 标准的格式说明符字符串

如下代码是格式字符串组件语法的一个示例：



如下代码给出了不同格式字符串的一些示例：

```
double myDouble = 12.345678;
Console.WriteLine("{0,-10:G} -- General",                    myDouble);
Console.WriteLine("{0,-10} -- Default, same as General",     myDouble);
Console.WriteLine("{0,-10:F4} -- Fixed Point, 4 dec places", myDouble);
Console.WriteLine("{0,-10:C} -- Currency",                   myDouble);
Console.WriteLine("{0,-10:E3} -- Sci. Notation, 3 dec places", myDouble);
Console.WriteLine("{0,-10:x} -- Hexadecimal integer",        1194719 );
```

这段代码产生了如下的输出：

```
12.345678  -- General
12.345678  -- Default, same as General
12.3457    -- Fixed Point, 4 dec places
$12.35     -- Currency
1.235E+001 -- Sci. Notation, 3 dec places
123adf     -- Hexadecimal integer
```

### 3. 标准数字格式说明符

Windows控制面板的区域和语言选项程序会影响一些说明符的最终格式。例如，指定国家或区域的货币符号会被货币格式说明符所使用。

表25-2总结了9种标准数字格式说明符。第一列在说明符名后列出了说明符字符。如果说明符字符根据它们的大小写会有不同的输出，就会标注要区分大小写。

表25-2　标准数字格式说明符

| 名字和字符 | 意　义 |
|---|---|
| 货币<br>C、c | 使用货币符号把值格式化为货币<br>精度说明符：小数位数<br>示例：Console.WriteLine("{0:C}",12.5);输出：$12.50 |
| 十进制数<br>D、d | 十进制数字字符串，需要的情况下有负数符号。只能和整数类型配合使用<br>精度说明符：输出字符串中的最少位数。如果实际数字的位数更少，则在左边以0填充<br>示例：Console.WriteLine("{0:D4}",12);<br>输出：0012 |
| 定点<br>F、f | 带有小数点的十进制数字字符串。如果需要也可以有负数符号<br>精度说明符：小数的位数<br>示例：Console.WriteLine("{0:F4}",12.3456789);<br>输出：12.3457 |
| 常规<br>G、g | 在没有指定说明符的情况下，会根据值转换为定点或科学记数法表示的紧凑形式<br>精度说明符：根据值<br>示例：Console.WriteLine("{o.G4}",12.345678);<br>输出：12.35 |
| 十六进制数<br>X、x<br>区分大小写 | 十六进制数字的字符串。十六进制数字A～F会匹配说明符的大小写形式<br>精度说明符：输出字符串中的最少位数。如果实际数的位数更少，则在左边以0填充<br>示例：Console.WriteLine("{0:x}",180026);<br>输出：2bf3a |

（续）

| 名字和字符 | 意　义 |
| --- | --- |
| 数字<br>N、n | 和定点表示法相似，但是在每三个数字的一组中间有分隔符。从小数点开始往左数<br>精度说明符：小数的位数<br>示例：Console.WriteLine("{0:N2}",12345678.54321);<br>输出：12,345,678.54 |
| 百分比<br>P、p | 表示百分比的字符串。数字会乘以100<br>精度说明符：小数的位数<br>示例：Console.WriteLine("{0:P2}",0.1221897);<br>输出：12.22% |
| 往返过程<br>R、r | 保证输出字符串后如果使用Parse方法将字符串转化成数字，那么该值和原始值一样<br>精度说明符：忽略<br>示例：Console.WriteLine("{0:R}",1234.21897);<br>输出：1234.21897 |
| 科学记数法<br>E、e<br>区分大小写 | 具有尾数和指数的科学记数法。指数前面加字母*E*。*E*的大小写和说明符一致<br>精度说明符：小数的位数<br>示例：Console.WriteLine("{0:e4}",12.3456789);<br>输出：1.2346e+001 |

## 25.3 把字符串解析为数据值

字符串都是Unicode字符的数组。例如，字符串"25.873"是6个字符而不是一个数字。尽管它看上去像数字，但是我们不能对它使用数学函数。把两个字符串进行“相加”只会串联它们。

- 解析允许我们接受表示值的字符串，并且把它转换为实际值。
- 所有预定义的简单类型都有一个叫做Parse的静态方法，它接受一个表示这个类型的字符串值，并且把它转换为类型的实际值。

以下语句给出了一个使用Parse方法语法的示例。注意，Parse是静态的，所以我们需要通过目标类型名来调用它。

```
double d1 = double.Parse("25.873");
                ↑            ↑
             目标类型     要转换的字符串
```

以下代码给出了一个把两个字符串解析为double型值并把它们相加的示例：

```
static void Main()
{
   string s1 = "25.873";
   string s2 = "36.240";

   double d1 = double.Parse(s1);
   double d2 = double.Parse(s2);

   double total = d1 + d2;
```

```
    Console.WriteLine("Total:  {0}", total);
}
```

这段代码产生了如下的输出：

```
Total:  62.113
```

如果字符串不能被解析，系统会抛出一个异常。还有另外一个静态方法TryParse，如果字符串解析成功，则返回true，其余情况返回false。如果解析失败，它不会抛出异常。

**说明**　关于Parse有一个常见的误解，由于它是在操作字符串，会被认为是string类的成员。其实不是，Parse不是一个方法，而是由目标类型实现的很多个方法。

## 25.4　可空类型

在某些情况下，特别是在使用数据库时，我们可能会希望表示一个当前没有保存有效值的变量。对于引用类型，这很容易实现，只要把变量设置为null就可以了。然而，如果我们的变量定义为值类型，不管它的内容是否是有效的含义，内存都已经分配了。

在这种情况下，我们需要一个关联到变量的布尔指示器，当值是有效时，指示器是true；而当值是无效时，指示器是falae。

C# 2.0引入的可空类型允许我们创建一个值类型的变量并标注它为有效或无效，这样，我们就可以在使用它之前确保变量是有效的。普通值类型称作不可空（non-nullable）类型。

### 25.4.1　创建可空类型

可空类型总是基于另外一个叫做基础类型（underlying type）的已经被声明的类型。

- 可以从任何值类型创建可空类型，包括预定义的简单类型。
- 不能从引用类型或其他可空类型创建可空类型。
- 不能在代码中显式声明可空类型，只能声明可空类型的变量。之后我们会看到，编译器会使用泛型隐式地创建可空类型。

要创建可空类型的变量，只需要在变量声明中的基础类型的名字后面加一个问号。不幸的是，这种语法看上去好像你的代码有很多问题一样。

例如，以下代码声明了一个可空int类型的变量。注意，后缀附加到类型名——而不是变量名称。

```
  后缀
   ↓
int? myNInt = 28;
 ↑
可空类型的名字包含后缀
```

有了这样的声明语句，编译器就会产生可空类型并关联变量类型。可空类型的结构如图 25-3 所示。它包含如下：

- ❑ 基础类型的实例。
- ❑ 几个重要的只读属性。
  - ■ HasValue属性是bool类型，并且指示值是否有效。
  - ■ Value属性是和基础类型相同的类型并且返回变量的值——如果变量是有效的话。

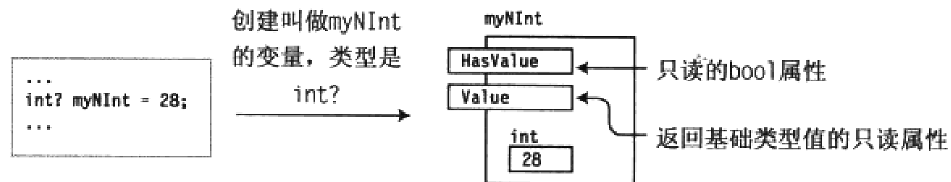



图25-3 可空类型在一个结构中包含了基础类型对象，并且还有两个属性

我们可以按如下方式显式使用两个只读属性：

```
int? myInt1 = 15;
        显式使用属性
            ↓
if ( myInt1.HasValue )
   Console.WriteLine("{0}", myInt1.Value);
                                ↑
                            显式使用属性
```

然而，更好的方式是使用快捷形式，如以下代码所示。

- ❑ 要检测可空类型是否有值，我们可以把它和null比较。
- ❑ 和其他变量一样，要获取它的值，可以只用它的名字。

```
        和null比较
            ↓
if ( myInt1 != null )
   Console.WriteLine("{0}", myInt1);
                              ↑
                          使用变量名
```

两组代码都产生如下的输出：

```
15
```

读取可空类型的变量会返回它的值。然而，你必须确保变量不是null。尝试读取null值的变量会产生一个异常。

我们可以简单地在可空类型和它对应的不可空类型之间转换。

- ❑ 不可空类型到它的可空形式之间有隐式转换。也就是说，不需要强制转换。
- ❑ 可空类型到它的不可空形式之间可以显式转换。

例如，以下几行代码演示了双向的转换。在第一行中，int类型被隐式转换为int?类型的值，并且被用于初始化可空类型的变量。在第二行中，变量被显式转换为它的不可空形式。

```
int? myInt1 = 15;                        // Implicitly convert int to int?
int  regInt = (int) myInt1;              // Explicitly convert int? to int
```

## 25.4.2 为可空类型赋值

有三种方式可以为可空类型的变量赋值：

- 基础类型的值
- 同样可空类型的值
- 值null

以下代码给出了三种赋值方式的示例：

```
int? myI1, myI2, myI3;

myI1 = 28;                                // 基础类型的值
myI2 = myI1;                              // 可空类型的值
myI3 = null;                              // 空

Console.WriteLine("myI1: {0}, myI2: {1}", myI1, myI2);
Console.WriteLine("myI3 {0} null", myI3 == null ? "is" : "is not");
```

这段代码产生了如下的输出：

```
myI1: 28, myI2: 28
myI3 is null
```

**使用运算符以及空接合运算符**

标准算术运算符和比较运算符同样也能处理可空类型。还有一个新的运算符叫做空接合运算符（null coalescing operator），它允许我们在可空类型变量为null时返回一个值给表达式。

空接合运算符由两个连续的问号组成，它有两个操作数：

- 第一个操作数是可空类型的变量。
- 第二个是相同基础类型的不可空值。
- 在运行时，如果第一个操作数运算后为null，那么第二个操作数就会被返回作为运算结果。

```
                                空接合运算符
                                     ↓
int? myI4 = null;
Console.WriteLine("myI4: {0}", myI4 ?? -1);

myI4 = 10;
Console.WriteLine("myI4: {0}", myI4 ?? -1);
```

这段代码产生了如下的输出：

```
myI4: -1
myI4: 10
```

对于相等比较运算符==和!=，我们需要知道其中的一个有趣的特性。如果你比较两个相同可空类型的值，并且都设置为null，那么相等比较运算符会认为它们是相等的。例如，在下面的代码中，两个可空的int被设置为null，相等比较运算符会声明它们是相等的。

```
int? i1 = null, i2 = null;                    // 都为空

if (i1 == i2)                                 // 返回true
   Console.WriteLine("Equal");
```

### 25.4.3 使用可空用户自定义类型

至此，我们已经看到了预定义的简单类型的可空形式。我们还可以创建用户自定义值类型的可空形式。这就引出了在使用简单类型时候没有遇到的其他问题。

主要问题是访问封装的基础类型的成员。一个可空类型不直接暴露基础类型的任何成员。例如，来看看下面的代码和图25-4中它的表示形式。代码声明了一个叫做MyStruct的结构（值类型），它有两个公共字段。

- 由于结构的字段是公共的，所以它可以被结构的任何实例所访问到，如图左部分所示。
- 然而，结构的可空形式只通过Value属性暴露基础类型，它不直接暴露它的任何成员。尽管这些成员对结构来说是公共的，但是它们对可空类型来说不是公共的，如图25-4右部分所示。

```
struct MyStruct
{
   public int X;
   public int Y;
   public MyStruct(int xVal, int yVal)
   { X = xVal;  Y = yVal; }
}

class Program {
   static void Main()
   {
      MyStruct? mSNull = new MyStruct(5, 10);
      ...
```

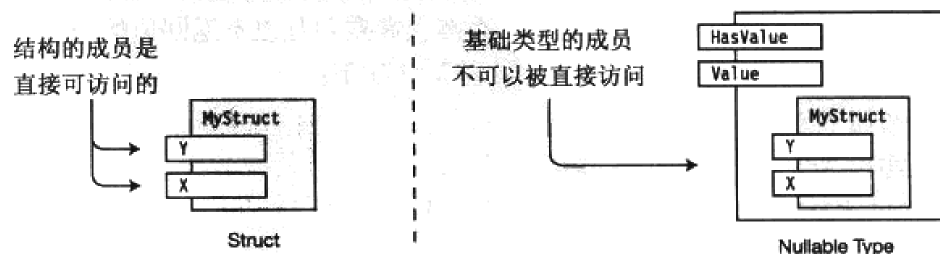



图25-4 结构成员的可访问性不同于可空类型

例如，以下代码使用之前声明的结构并创建了结构和它对应的可空类型的变量。在代码的第三行和第四行中，我们直接读取结构变量的值。在第五行和第六行中，就必须从可空类型的Value属性返回的值中进行读取。

```
MyStruct  mSStruct = new MyStruct(6, 11);       // 结构变量
MyStruct? mSNull   = new MyStruct(5, 10);       // 可空类型的变量
                                      结构访问
                                         ↓
Console.WriteLine("mSStruct.X: {0}", mSStruct.X);
Console.WriteLine("mSStruct.Y: {0}", mSStruct.Y);

Console.WriteLine("mSNull.X: {0}",   mSNull.Value.X);
Console.WriteLine("mSNull.Y: {0}",   mSNull.Value.Y);
                                          ↑
                                      可空类型访问
```

### Nullable<T>

可空类型通过一个叫做System.Nullable<T>的.NET类型来实现，它使用了C#的泛型特性。

C#可空类型的问号语法是创建Nullable<T>类型变量的快捷语法，在这里T就是基础类型。Nullable<T>接受了基础类型并把它嵌入结构中，同时给结构提供可空类型的属性、方法和构造函数。

我们可以使用Nullable<T>这种泛型语法，也可以使用C#的快捷语法。快捷语法更容易书写和理解，并且也减少了出错的可能性。

以下代码使用Nullable<T>语法为之前示例中声明的MyStruct结构创建一个叫做mSNull的Nullable<MyStruct>类型：

```
Nullable<MyStruct> mSNull = new Nullable<MyStruct>();
```

下面的代码使用了问号语法，完全等同于Nullable<T>语法：

```
MyStruct? mSNull = new MyStruct();
```

## 25.5 Main 方法

每一个C#程序都必须有一个入口点——一个必须叫做Main的方法。

在贯穿本书的示例代码中，都使用了一个不接受参数并且也不返回值的Main方法。然而，一共有4种形式的Main可以作为程序的入口点。这些形式如下：

- ❑ `static void Main                   {...}`
- ❑ `static void Main(string[] args)    {...}`
- ❑ `static int Main()                  {...}`
- ❑ `static int Main(string[] args)     {...}`

前面两种形式在程序终止后都不返回值给执行环境。后面两种形式则返回int值。如果使用返回值，通常用于报告程序的成功或失败，0通常用于表示成功。

第二种和第四种形式允许我们在程序启动时从命令行向程序传入实参，也叫做参数。命令行参数的一些重要特性如下：

- 可以有0个或多个命令行参数。即使没有参数，args参数也不会是null，而是一个没有元素的数组。
- 参数由空格或制表符隔开。
- 每一个参数都被程序解释为是字符串，但是你无须在命令行中为参数加上引号。

例如，下面叫做CommandLineArgs的程序接受了命令行参数并打印了每一个提供的参数：

```
class Program
{
   static void Main(string[] args)
   {
      foreach (string s in args)
         Console.WriteLine(s);
   }
}
```

如下命令行使用5个参数执行CommandLineArgs程序。

```
CommandLineArgs Jon Peter Beth Julia Tammi
      ↑                    ↑
  可执行程序名             参数
```

前面的程序和命令行产生了如下的输出：

```
Jon
Peter
Beth
Julia
Tammi
```

其他需要了解的有关Main的重要事项如下：

- Main必须声明为static。
- Main可以被声明为类或结构。

一个程序只可以包含Main的4种可用入口点形式中的一种声明。当然，如果你声明其他方法的名称为Main，只要它们不是4种入口点形式中的一种就是合法的——但是，这样做是非常容易混淆的。

**Main的可访问性**

Main可以被声明为public或private：

- 如果Main被声明为private，其他程序集就不能访问它，只有执行环境才能启动程序。
- 如果Main被声明为public，其他程序集就可以调用它。

然而，无论Main声明的访问级或所属类或结构的访问级别是什么，执行环境总是能访问Main。

默认情况下，当Visual Studio创建了一个项目时，它就创建了一个程序框，其中的Main是隐式private。如果需要，你随时可以添加public修饰符。

## 25.6 文档注释

文档注释特性允许我们以XML元素的形式在程序中包含文档。Visual Studio会帮助我们插入元素，以及从源文件中读取它们并拷贝到独立的XML文件中。本小节不介绍XML的主题，但是会介绍使用文档注释的大致过程。

图25-5给出了一个使用XML注释的概要。这包括如下步骤：

- 你可以使用Visual Studio来产生带有嵌入了XML的源文件。Visual Studio会自动插入大多数重要的XML元素。
- Visual Studio从源文件中读取XML并且拷贝XML代码到新的文件。
- 另外一个叫做文档编译器的程序可以获取XML文件并且从它产生各种类型的文件。

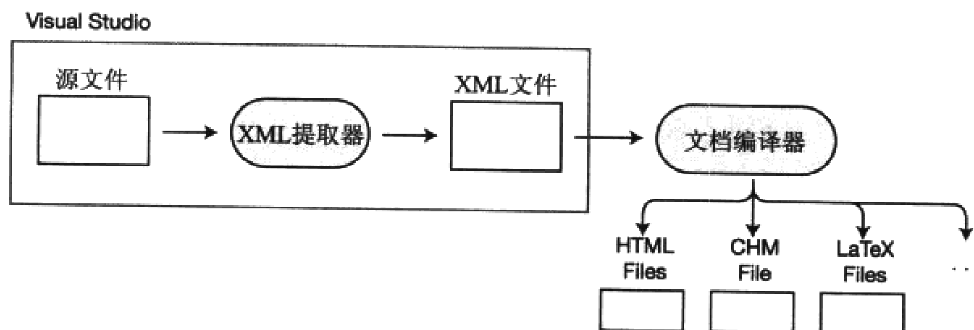



图25-5 XML注释过程

之前的Visual Studio版本包含了基本的文档编译器，但是它在Visual Studio 2005发布之前被删除了。微软公司正在开发一个叫做Sandcastle的新文档编译器，它已经被用来生成.NET框架的文档。你可以从“微软开发者网络”的网站（http://msdn.microsoft.com）上下载这个软件。

### 25.6.1 插入文档注释

文档注释从三个连续的正斜杠开始。

- 前两个斜杠指示编译器这是一行注释，并且需要从程序的解析中忽略。
- 第三个斜杠指示这是一个文档注释。

例如，以下码中前4行就是有关类定义的文档注释。这里使用<summary>XML标签。在字段声明之上有三行来说明这个字段——还是使用<summary>标签。

```
/// <summary>          ← 类的开始XML标签
/// This is class MyClass, which does the following wonderful things, using
/// the following algorithm. ... Besides those, it does these additional
```

```
/// wonderful things.
/// </summary>        ← 关闭XML标签
class MyClass
{
   /// <summary>      ← 字段的开始XML标签
   /// Field1 is used to hold the value of ...
   /// </summary>     ← 关闭XML标签
   public int Field1 = 10;
   ...
```

每一个XML元素都是当我们在语言特性（比如类或类成员）的声明上输入3条斜杠时，Visual Studio自动增加的。

例如，从下面的代码可以看到，在MyClass类声明之上的2条斜杠：

```
//
class MyClass
{ ...
```

只要我们增加了第三条斜杠，Visual Studio会立即扩展注释为下面的代码，而我们无须做任何事情。然后我们就可以在标签之间输入任何希望注释的行了。

```
/// <summary>        Automatically inserted
///                  Automatically inserted
/// </summary>       Automatically inserted
class MyClass
{ ...
```

### 25.6.2 使用其他 XML 标签

在之前的示例中，我们看到了summay XML标签的使用。C#可识别的标签还有很多。表25-3列出了最重要的一些。

表25-3 文档代码XML标签

| 标 签 | 意 义 |
| --- | --- |
| <code> | 格式化内部的行，用看上去像代码的字体 |
| <example> | 将内部的行标注为一个示例 |
| <param> | 为方法或构造函数标注参数，并允许描述 |
| <remarks> | 描述类型的声明 |
| <returns> | 描述返回值 |
| <seealso> | 在输出文档中创建SeeAlso一项 |
| <summary> | 描述类型或类型成员 |
| <value> | 描述属性 |

## 25.7 嵌套类型

我们通常直接在命名空间中声明类型。然而，我们还可以在类或结构中声明类型。

- 在另一个类型声明中声明的类型叫做嵌套类型。和所有类型声明一样，嵌套类型是类型实例的模板。
- 嵌套类型像封闭类型（enclosing type）的成员一样声明。
  - 嵌套类型可以是任意类型。
  - 嵌套类型可以是类或结构。

例如，以下代码显示了MyClass类，其中有一个叫做MyCounter的嵌套类。

```
class MyClass                          // 封闭类
{
   class MyCounter                     // 嵌套类
   {
      ...
   }
   ...
}
```

如果一个类型只是作为帮助方法并且只对封闭类型有意义，可能就需要声明嵌套类型了。

不要被嵌套这个术语弄混淆。嵌套指声明的位置——而不是任何实例的位置。尽管嵌套类型的声明在封闭类型的声明之内，但嵌套类型的对象并不一定封闭在封闭类型的对象之内。嵌套类型的对象——如果创建了的话——和它没有在另一个类型中声明时所在的位置一样。

例如，图25-6显示了前面代码框架中的MyClass对象和MyCounter对象。另外还显示了MyClass类中一个叫做Counter的字段，这就是指向嵌套类型对象的引用，它在堆的另一处。

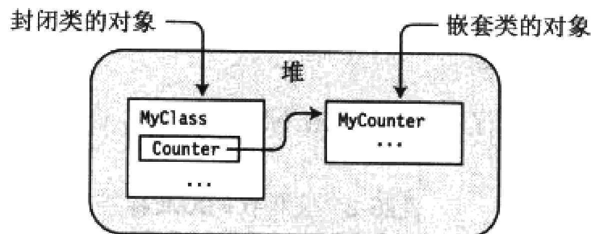



图25-6　嵌套指的是声明的位置而不是对象的位置

### 25.7.1　嵌套类的示例

以下代码把MyClass和MyCounter完善成了完整的程序。MyCounter实现了一个整数计数器，从0开始并且使用++运算符来递增。当MyClass的构造函数被调用时，它创建嵌套类的实例并且为字段分配引用。图25-7演示了代码中对象的结构。

```
class MyClass
{
   class MyCounter                              // 嵌套类
   {
      private int _Count = 0;
      public int Count                          // 只读属性
```

```
            {
               get { return _Count; }
            }

            public static MyCounter operator++( MyCounter current )
            {
               current._Count++;
               return current;
            }
         }

         private MyCounter counter;                          // 嵌套类的字段

         public MyClass() { counter = new MyCounter(); }    // 构造函数

         public int Incr()      { return (counter++).Count; }  // 增量方法
         public int GetValue() { return counter.Count; }      // 获取counter的值
      }

      class Program
      {
         static void Main( )
         {
            MyClass mc = new MyClass();

            mc.Incr(); mc.Incr(); mc.Incr();
            mc.Incr(); mc.Incr(); mc.Incr();

            Console.WriteLine("Total:  {0}", mc.GetValue());
         }
      }
```

这段代码产生了如下的输出：

```
Total:  6
```

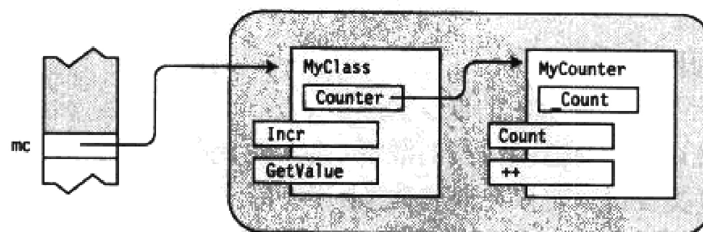



图25-7　嵌套类的对象以及它的封闭类

## 25.7.2 可见性和嵌套类型

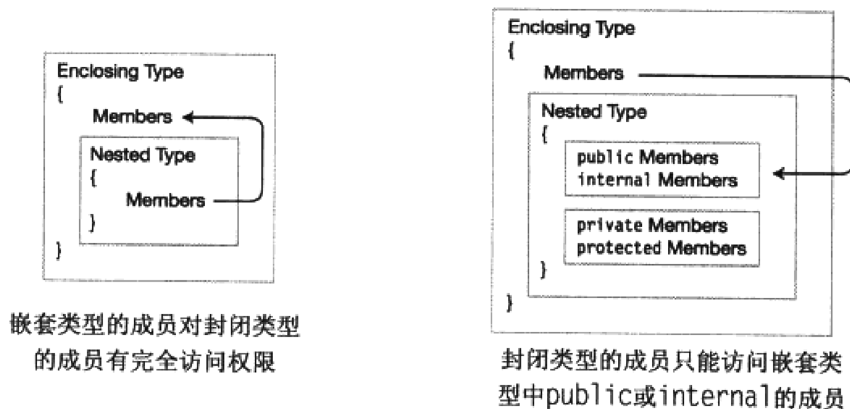
在第7章中，我们已经了解到类和类型通常有public或internal的访问级别。然而，嵌套类型的不同之处在于，它们有成员访问级别而不是类型访问级别。因此，下面的命题是成立的：

- 在类内部声明的嵌套类型可以有5种类成员访问级别中的任何一种：public、protected、private、internal或protected internal。
- 在结构内部声明的嵌套类型可以有3种结构成员访问级别中的任何一种：public、internal或private。

在这两种情况下，嵌套类型的默认访问级别都是private，也就是说不能被封闭类型以外的对象所见。

封闭类和嵌套类的成员之间的关系是很容易理解的，如图 25-8 所示。不管封闭类型的成员声明了怎样的访问级别，包括private和protected，嵌套类型都能访问这些成员。

然而，它们之间的关系不是对称的。尽管封闭类型的成员总是可见嵌套类型的声明并且能创建它的实例，但是它们不能完全访问嵌套类型的成员。相反，这种访问权限受限于嵌套类成员声明的访问级别——就好像嵌套类型是一个独立的类型一样。也就是说，它们可以访问public或internal的成员，但是不能访问嵌套类型的private或protected成员。




嵌套类型的成员对封闭类型的成员有完全访问权限

封闭类型的成员只能访问嵌套类型中public或internal的成员

图25-8　嵌套类型成员和封闭类型成员之间的可访问性

我们可以把这种关系总结如下。

- 嵌套类型的成员对封闭类型的成员总是有完全访问权限。
- 封闭类型的成员：
    - 总是可以访问嵌套类型本身；
    - 只能访问声明了有访问权限的嵌套类型成员。

嵌套类型的可见性还会影响基类成员的继承。如果封闭类型有子类，嵌套类型就可以通过使用相同的名字来隐藏基类成员。可以在嵌套类型的声明上使用new修饰符来显式隐藏。

嵌套类型中的this引用指的是嵌套类型的对象——不是封闭类型的对象。如果嵌套类型的对象需要访问封闭类型，它必须持有封闭类型的引用。如以下代码所示，我们可以把封闭对象提供的this引用作为参数传给嵌套类型的构造函数：

```
class SomeClass                          // 封闭类
{
   int Field1 = 15, Field2 = 20;         // 封闭类的字段
```

```
    MyNested mn = null;                        // 嵌套类的引用

    public void PrintMyMembers()
    {
       mn.PrintOuterMembers();                 // 调用嵌套类中的方法
    }

    public SomeClass()
    {
       mn = new MyNested(this);                // 创建嵌套类的实例
    }                      ↑
                  给封闭类型传递引用
    class MyNested                             // 嵌套类声明
    {
       SomeClass sc = null;                    // 封闭类的引用

       public MyNested(SomeClass SC)           // 嵌套类的构造函数
       { olm
          sc = SC;                             // 存储嵌套类的字段
       }

       public void PrintOuterMembers()
       {
          Console.WriteLine("Field1: {0}", sc.Field1);
          Console.WriteLine("Field2: {0}", sc.Field2);
       }
    }                                          // 嵌套类结束
}

class Program {
   static void Main( ) {
      SomeClass MySC = new SomeClass();
      MySC.PrintMyMembers();
   }
}
```

这段代码产生了如下的输出：

```
Field1: 15
Field2: 20
```